Store Buffer Reduction with MMUs: Complete Paper-and-pencil Proof

Geng Chen[1,2], Ernie Cohen, and Mikhail Kovalev[1]

[1] Saarland University, Saarbrücken, Germany
{gengchen, kovalev}@wjpserver.cs.uni-saarland.de
[2] University of Electronic Science and Technology of China, Chengdu, China

Abstract. A fundamental problem in concurrent system design is to identify flexible programming disciplines under which weak memory models provide sequential consistency. For x86-TSO, a suitable reduction theorem for threads that communicate only through shared memory was given by Cohen and Schirmer [5]. However, this theorem cannot handle programs that edit their own page tables (e.g. memory mangers, hypervisors, and some device drivers). The problem lies in the interaction between a program thread and the hardware MMU that provides its address translation: the MMU cannot be treated as a separate thread (since it implicitly communicates with the program thread), nor as part of the program thread itself (since MMU reads do not snoop the store buffer of the program thread). We generalize the Cohen-Schirmer reduction theorem to handle programs that edit their page tables. The added conditions prevent the MMU of a thread from walking page table entries owned by other threads.

Keywords: Store buffer reduction, MMU, TLB, sequential consistency, verification

1 Introduction

When reasoning about concurrent software, programmers typically assume an "interleaving" model of concurrency, formalized as sequentially consistency (SC) [8], where all memory accesses appear to be linearly ordered. However, providing SC in hardware is relatively expensive, so modern multicore processors typically implement weaker memory models, where different threads can see writes appear in different orders. To implement programming languages that guarantee SC (such as Java), or to use SC reasoning about low-level programs written to execute directly on such hardware, we need practical program criteria that guarantee that any execution is guaranteed to simulate a SC one.

In this paper, we consider one of the more prevalent non-SC memory models, x86-TSO [9], which is essentially the native memory model provided by x86/x64 family processors. In x86-TSO, when a processor retires an instruction, its stores are queued in a FIFO *store buffer* (SB); only when such a store emerges from

the store buffer is it applied to the global, shared memory, and made visible to other processors. To make SBs transparent to single-threaded programs, when a processor issues load for of addresses, it first checks whether there is a store to that address in its SB. If there is such a buffered store, it uses the most recent one to satisfy the load; otherwise, it loads the value from memory. Thus, a store from a processor becomes visible to (later) loads from that processor as soon as it enters the SB, and so becomes visible to the thread issuing the store (and other threads running on that processor) before it becomes visible to threads running on other processors. Thus, stores can appear in different order for different threads, violating SC. For example:

```
T1: a1:=1                          T2: a2:=1
    if(a2==0)                          if(a1==0)
      critical section                   critical section
```

In a SC execution, it is impossible for both threads to enter the critical section. But under x86-TSO, both comparisons might succeed before the stores `a1` and `a2` emerge from their respective SBs, allowing both threads to enter the critical section.

One way to reason about programs running on x86-TSO is to simply materialize the SBs explicitly in the programming model. This approach is hopelessly impractical; for example, the postcondition of a function call would have to explicitly talk about the sequence of stores left in the SB, exposing the internal details of the function implementation to its callers and destroying modular program reasoning.

A discipline guaranteeing sequential consistency, like the one presented in this paper, would disallow a program such as the on above; typically, it would require a thread to flush its store buffer at some point between a volatile store (like the assignment to `a1` above) and a subsequent volatile load (like the read of `a2`). A store buffer reduction theorem, giving such a discipline for x86-TSO, was given by Cohen and Schirmer in [5]. The main challenge in making such a discipline practical is avoiding introducing any flushing obligations for accesses that do not race with other threads; the reduction theorem achieves this by means of an ownership discipline, where conformance to the ownership discipline is itself verified assuming SC.

A substantial complication arises when we extend the model to include virtual address translation. (Such translation is invisible to most user-space programs, but is visible to programs that edit their own page tables, such as memory managers and hypervisors.) Since the MMU is naturally modelled as a separate thread, it is tempting to try to apply the store buffer reduction theorem directly. The problem is that the store buffer reduction theorem taxitly assumes the processors communicate only through shared memory, whereas a processor and its MMU implicitly share address translations cached in the Translation Lookaside Buffer (TLB). Moreover, we cannot simply treat the TLB as volatile shared memory, since the discipline would require the processor to flush its store buffer between each of its writes to page tables and the following (implicit) read of the

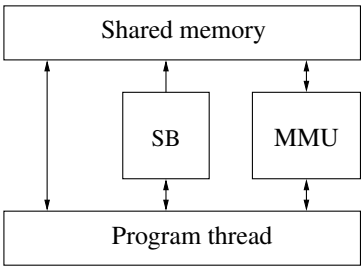

Fig. 1: Abstract view of x86-TSO with the address translation.

TLB; these additional flushes would essentially force a flush after each write to a page table, rendering the discipline impractical. The purpose of this paper is to extend the reduction theorem to accommodate SBs, without introducing such flushes.

The rest of this paper is structured as follows. In Sec. 2 we briefly describe the MMU behaviour in the x86-TSO execution. In Sect. 3 we give the informal description of our programming discipline. In Sect. 4 we define the virtual machine with sequentially consistent memory and the SB machine and formalize the programming discipline defining safety conditions for the virtual machine. In Sect. 5 we state the reduction theorem and give the intuition behind the proof, as well as the overall proof structure. There we also define the coupling relation between the machines, state a refined version of the safety criteria (called safety of the delayed release) and define auxiliary invariants for the SB machine. In Sect. 6 we show that the invariants are maintained during the SB machine execution. In Sect. 7 we prove simulation using safety of the delayed release and in Sect. 8 we derive safety of the delayed release from the regular safety of the virtual machine. In Sect. 9 we conclude and give a short summary of the future work.

2 MMUs

With the presence of address translation, every thread communicates with a user-visible MMU component (Fig. 1). We assume the TLB to be a part of the MMU state. The MMU component can perform non-deterministic steps fetching a page table entry (PTE) from the memory or writing the memory (setting control bits in a PTE). Every read of a PTE can change the state of the MMU, extending the set of translations cached in a TLB. When a thread is running in translated mode, a memory access can be executed only when the MMU can provide a suitable address translation for the virtual address of the access. Exact state of the MMU is never known to the user, because MMUs are allowed to perform speculative address translations and to cache them in the TLBs. Page tables can be either thread-local or shared between different MMUs.

In the concurrent execution one can consider an MMU as another thread executing memory accesses, which differs from a regular thread in several ways: (i) MMU does not have its own SB, (ii) MMU steps generally cannot be reordered behind the steps of the regular thread, because those steps might depend on the address translations obtained from the MMU and (iii) for a given program there might be many possible sequences of MMU steps due to the non-deterministic nature of the MMU and the ability of the MMU to cache the complete and incomplete address translations in the TLB.

A thread can also perform reads and writes to page tables, changing the superset of possible address translations for its own MMU and for MMUs of other threads on the fly. Note though, that MMUs are allowed to cache the "old" address translations in TLBs. Hence, when overwriting an address translation in page tables one has to make sure that proper TLB invalidation is performed before executing an instruction, which can request the overwritten address translation.

To illustrates the possible violations of sequential consistency with the presence of an MMU (even in a single threaded environment) we consider the following example.

```
T1: pte2.p:=0          MMU1: pte1.a:=1
    t0:=pte1.a                t1:= pte2
```

Let in the initial state `pte1` and `pte2` be thread-local page table entries, where `pte1` points to `pte2`, present bits `p` in both entries are set, and access bit `pte1.a` equals 0. Consider a TSO execution where the steps of the thread are executed before the steps of the MMU and the write to the present bit of `pte2` is put to an SB. After this execution `t0` equals 0 and the MMU reads `pte2` where the present bit is set. As a result, the MMU gets an address translation which goes through `pte1` and `pte2`. In a sequentially consistent execution the MMU would read `pte2` with the low present bit, which means that no address translation through this entry can be provided (possibly a page fault would be raised). We also cannot move the MMU steps in front of the read to `t0`, because `t0` would then get 1. In general, we are allowed to reorder the MMU read alone in front of the MMU write to `pte1` and in front of the read to `t0`, but for the example given above this is not feasible, because MMU is allowed to read `pte2` only after the access bit in `pte1` is set.

3 Programming Discipline

The programming discipline introduced here is an extension of the programming discipline from [5] and is based on ownership sets, which have to be maintained explicitly by the user with the help of the ghost code. All memory accesses can be either shared (volatile) or local and must be *safe* i.e., obey the rules of the programming discipline. Semantically there is no difference between both types of the accesses, but we enforce different rules on volatile and non-volatile memory

operations. The interlocked accesses, i.e. those accesses which flush the SB as a side effect, follow the same rules as volatile accesses.

Ownership information is maintained in the ghost state. We distinguish between the following ownership sets of memory addresses:

- Shared, unowned read-write addresses: used for implementing locks [6], lock-free algorithms or shared page tables. Every thread can perform volatile reads and writes to these addresses and MMU of every thread is allowed to read and write this memory.
- Shared, unowned read-only addresses: used for static data. Every thread can perform volatile and non-volatile reads from these addresses.
- Shared, owned read-write addresses: used for single-writer-multiple-readers data structures. Every thread can perform volatile reads, but only the (unique) owner is allowed to do volatile writes to these addresses.
- Unshared, owned read-write addresses: used for thread-local data or for data protected by a lock. The owner is allowed to write and read the data with volatile and non-volatile accesses.
- Owned page table addresses: used for local page tables. The owning thread is allowed to read and write these addresses with volatile accesses. The MMU of the owning thread is allowed to read and write this memory.

Note, that the set of addresses which can be accessed by the MMU of a thread is actually defined by the set of reachable PTEs from the address of the root page table, which is stored in a register. Hence, our discipline requires every reachable PTE address to be either in the set of local page table addresses or to be in the set of shared, unowned read-write addresses. The latter is useful in situations when several concurrent threads are sharing the same set of page tables for address translation. Moreover, a local page table can point to a page table shared by MMUs of several threads, which allows to split the address space of a thread into local and shared parts. The other direction is also possible, i.e. a page table located in the shared memory can contain a link to a local page table. In this case any thread can write the shared page table, but only the MMU of the thread which owns the local page table should be able to access both of them. If another MMU would have an access to the "shared" page table, it would automatically be able to access all page tables linked to it, which violates the safety of the MMU access.

Ownership can be transfered either by a non-blocking ghost update or as a side effect of a volatile or interlocked write operation. The latter is helpful e.g. when one acquires a lock and wants to get the ownership of the memory protected by the lock. A thread is allowed to acquire ownership of an unowned address and to release the ownership of an owned address. When a thread acquires an unowned address, it can either make it owned unshared, owned shared or an owned page table address. When releasing an owned address a thread decides whether to make it shared read-write or shared read-only. It also can make a shared address which it already owns unshared.

The flushing rule of our programming discipline stays unchanged from [5]: an SB has to be flushed before every volatile read if this read was preceded by a

volatile write. This guarantees, that the thread always makes its updates of the shared state visible to others, before it reads a shared variable. Local page tables in this sense are considered as shared state between a thread and its MMU. A *dirty flag* for every thread is maintained in the ghost state in order to identify if the thread has performed a volatile write since the last SB flush. Safety of a volatile read makes sure that the read is performed only when the dirty flag is not set. Note, that the rules of the programming discipline described above can be checked in a sequentially consistent context with the ghost state, e.g. in a software verifier like VCC [3].

We reconsider the example with a thread and an MMU from Sect. 1.

```
T1: assert(ownedpt(pte2))                MMU1: pte1.a:=1
    vol pte2.p:=0 {D:=1}                       t1:= pte2
    FENCE {D:=0}
    assert(ownedpt(pte1) && D==0)
    vol t0:= pte1.a
```

Before accesses `pte1` and `pte2` we assert that both PTEs are present in the local page table set of the thread (alternatively, we could consider page table entries, which are present in the shared unowned set of addresses). The write to `pte2` has to be volatile, which means that the dirty bit `D` of the thread is set to 1. We update the dirty bit with a ghost operation, which is represented with $\{\dots\}$ and denote volatile operations with `vol`. A read of `pte1` also has to be volatile. As a precondition of volatile reads we requires the dirty bit to be equal 0. Hence, in between a write and a read we insert a fence, the only effect of which is a store buffer flush resulting in a dirty bit to be set to 0.

However, adding these annotations to the code of the program is not enough. We also have to make sure that the MMU can only perform safe accesses, i.e. accesses to the local and shared page tables. One way to check this is to explicitly model the MMU in the ghost state of the program and to introduce a ghost `T1'` thread, which non-deterministically executes all possible MMU steps. By verifying such a program in VCC one can make sure that MMU accesses are always safe [2].

```
T1: assert(ownedpt(pte2))                T1': ...
    vol pte2.p:=0 {D:=1}                      assert(ownedpt(pte1))
    FENCE {D:=0}                              {pte1.a:=1}
    assert(ownedpt(pte1) && D==0)             assert(ownedpt(pte2))
    vol t0:= pte1.a                           {t1:= pte2}
                                              ...
```

Note, that we require the translated physical addresses, rather than the untranslated virtual addresses, to adhere to our programming discipline. Showing that the translated physical addresses of memory accesses are safe can be done if one keeps track of the set of all possible address translations for a given thread. This set can be maintained as part of the MMU state in the ghost state of the program [2].

4 Formalization

The set of natural numbers is denoted by $\mathbb{N}$ and the set of boolean values by $\mathbb{B}$. We denote the set of (both virtual and physical) memory addresses by $\mathbb{A}$, the set of values by $\mathbb{V}$ and the set of read temporaries by $\mathbb{T}$. A memory is modeled as a mapping from $\mathbb{A}$ to $\mathbb{V}$. A map update is written as $m(a \mapsto v)$. We restrict the domain of a map m to a set A by $m \restriction_A$. A record update is denoted as $c[X := v]$. We also use nested record updates, e.g. $c[c_1.X := v]$. We denote the length of a list l as $|l|$ and index lists from right to left i.e., $l[0 : |l| - 1]$. The n-th element of list l can be selected with $l[n]$ or $l_{[n]}$ (we use the latter notation to index components of thread n). Concatenation of two lists is denoted by $x \circ y$.

4.1 MMU Abstraction

Instead of considering a detailed x86 MMU model in the style of [2], we use a more abstract MMU model here. The set of MMU configurations we denote by $\mathbb{U}$. The MMU state also subsumes the TLB state and the current value of the page table origin register ($CR3$ register in x86). The set of all possible access rights is denoted by $\mathbb{R}$. A single page table entry occupies a single cell in the memory and has the same type $\mathbb{V}$ as all other memory values. Our MMU model relies on the following (uninterpreted) functions:

- $atran(mmu, va, mode, r) \in 2^{\mathbb{A}}$. Given an MMU state $mmu \in \mathbb{U}$, a virtual address $a \in \mathbb{A}$, translation mode $mode \in \mathbb{B}$ (1 - translated mode, 0 - untranslated mode) and the set of access rights $r \in \mathbb{R}$, the function returns the set of translated physical addresses for the specified access. In case there are no available translations the returned set is empty. For the untranslated mode function $atran$ should return $\{va\}$. We use this function to obtain an address translation when an instruction is being executed.
- $can\text{-}access(mmu, pa) \in \mathbb{B}$. For a physical address $pa \in \mathbb{A}$ the predicate denotes that the MMU can perform an access to a PTE located at address pa. This is the case when the MMU has fetched or has found in the TLB a valid PTE, which has the access and the present bits set and which points to the PTE located at address pa or when pa belongs to the top-level page table. We use this predicate as a guard for MMU read and MMU write steps.
- $\delta_{mmur}(mmu, pa, pte) \in 2^{\mathbb{U}}$. For page table entry $pte \in \mathbb{V}$ located at address pa the function returns the possible set of MMU states after the MMU has processed pte. After this step MMU can have complete or incomplete translations through pte buffered in the TLB. We use this function to obtain the new state of the MMU after the MMU read step.
- $\delta_{mmuw}(mmu, pa, pte) \in 2^{\mathbb{V}}$. This function returns the set of possible PTE values which can be written by the MMU at address pa, given that pte is the current value of the PTE located at address pa. This step models setting of access and dirty bits in a page table entry. We use this function when performing an MMU write step.

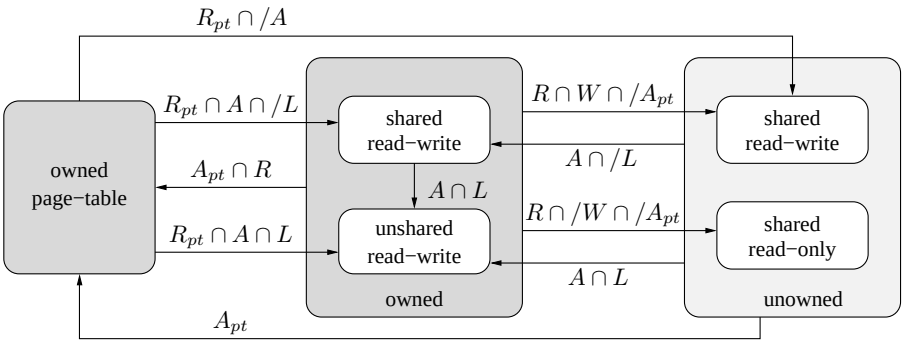

Fig. 2: Ownership transfer.

- $can\text{-}page\text{-}fault(mmu, va, r, pa, pte) \in \mathbb{B}$. This predicate denotes that the MMU can signal a page fault for the virtual address va and access rights r. The condition for the page fault [3] must be present in the page table entry pte located at address pa and the MMU must already have an incomplete address translation leading to pte buffered in the TLB.
- $\delta_{flush}(mmu, F, fi) \in \mathbb{U}$. For the set of (virtual) addresses $F \in 2^{\mathbb{A}}$ and the flag $fi \in \mathbb{B}$ the function performs a TLB flush, removing translations for addresses in F from the TLB, and returns the new MMU state after the flush is performed. The flag fi denotes that incomplete translations for addresses not in F also must be flushed.
- $\delta_{wpto}(mmu, v) \in \mathbb{U}$. The function performs a complete SB flush and sets the new value $v \in \mathbb{V}$ for the page table origin (PTO).

We assume *monotonicity* of the MMU i.e., after MMU performs a read of a PTE its set of address translations which can be provided by the MMU can only grow:

$$atran(mmu, va, mode, r) \subseteq atran(\delta_{mmur}(mmu, pa, pte), va, mode, r).$$

4.2 Instructions

The ghost ownership annotations consist of the following sets of addresses: acquired addresses A, the local fraction of acquired addresses L, released addresses R, the writable fraction of released addresses W, acquired page table addresses A_{pt} and released page table addresses R_{pt}. The ownership transfer is performed by write, ghost and read-modify-write (RMW) instructions. The possible effect of the ownership transfer is given in Fig. 2.

[3] The page fault can be signalled if the present bit in a PTE is not set, or there is an access rights violation or some of the reserved validity bits in a PTE have wrong values.

The set of memory instructions $\mathbb{I}$ is defined as a datatype with the following constructors:

$$\begin{aligned}\mathbb{I} = \ & \textbf{Read} \text{ vol va t r} \mid \textbf{Write} \text{ vol va (D, f) r annot} \mid \textbf{Ghost} \text{ annot} \\ & \mid \textbf{RMW} \text{ va t (D, f) r cond annot} \mid \textbf{Fence} \mid \textbf{Switch} \text{ mode} \\ & \mid \textbf{INVLPG} \text{ F fi} \mid \textbf{WritePTO} \text{ v}\end{aligned}$$

Parameter *vol* denotes whether the memory access is volatile. *annot* is a tuple consisting of the ownership annotations $(A, L, R, W, A_{pt}, R_{pt})$. Instruction **Read** loads the value from virtual address *va* into temporary t. r denotes the set of access permissions, which will be used for the address translation of *va*. **Write** stores the value computed by function f at the virtual memory address *va*. Function f takes as a parameter the map from temporaries to values and returns a value. D specifies the set of temporaries on which function f operates. If the volatile flag is set then the instruction also performs the ownership transfer according to *annot*. The only effect of instruction **Ghost** is the ownership transfer. Instruction **RMW** first loads the value from virtual address *va* into temporary t. Then it computes the value of the predicate *cond* on the updated set of temporaries and performs a write to *va* together with the ownership transfer if the test succeeds. The value to be stored is calculated the same way as for instruction **Write**. Instruction **Fence** flushes the SB when executed by SB machine. Instruction **Switch** changes the translation mode to $mode \in \mathbb{B}$. **INVLPG** removes translations for the virtual addresses in F from the TLB. Flag fi denotes that we have to flush incomplete translations for all other virtual addresses as well. Instruction **WritePTO** updates the page table origin with value v. When executed by the SB machine instructions **RMW**, **Switch**, **INVLPG** and **WritePTO** also flush the SB as a side effect.

To distinguish between different kinds of instructions we introduce predicates $R(I)$, $W(I)$, $G(I)$, $RMW(I)$ - for read, write, ghost and RMW instructions respectively and $FENCE(I)$, $SWITCH(I)$, $INVLPG(I)$, $WPTO(I)$ - for fence, mode switch, INVLPG and write to PTO instructions. Volatile and non-volatile reads and writes are distinguished by predicates $vR(I)$, $vW(I)$ and $nvR(I)$, $nvW(I)$ respectively. Individual fields X of instruction I we denote by $I.X$.

4.3 Virtual Machine

The virtual machine is an abstract machine with sequentially consistent memory and with address translations. It does not have SBs. The virtual machine maintains additional ghost information which allows to enforce the ownership-based programming discipline both for instructions and for MMU memory accesses. We call an execution which maintains this programming discipline *safe*. The virtual machine also contains the ghost *release sets*, which accumulate history information about the addresses released by ghost instructions. These sets do not influence the execution of the machine in any way and are not used to specify the programming discipline. Hence, one can simply omit them when instantiating the virtual machine. In Sect. 5.3 we use these sets to get a refined version of the safety criteria.

Configuration. Configuration of the virtual machine c is defined as a tuple:

$$c = (m, shared, ro, ts) \in \mathbb{M},$$

where $ts \in [0 : np - 1] \to \mathbb{K}$ is the list of thread-local configurations of thread i, $m \in \mathbb{A} \to \mathbb{V}$ is the shared memory of the machine, $shared \in 2^{\mathbb{A}}$ is the (ghost) set of shared addresses and $ro \in 2^{\mathbb{A}}$ is the (ghost) set of read-only addresses.

Thread-local configuration $c.ts_{[i]}$ of thread i is defined as

$$c.ts_{[i]} = (p, is, \vartheta, mmu, \mathcal{D}, \mathcal{O}, pt, mode, rls_l, rls_s, rls_{pt}) \in \mathbb{K},$$

where $p \in \mathbb{P}$ is the (uninterpreted) program state of the thread, $is \in \mathbb{I}^*$ is the instruction list, $\vartheta \in \mathbb{T} \rightharpoonup \mathbb{V}$ is the set of read temporaries (a read buffer), $mmu \in \mathbb{U}$ is the MMU state, $\mathcal{D} \in \mathbb{B}$ is the (ghost) dirty flag, $\mathcal{O} \in 2^{\mathbb{A}}$ is the (ghost) thread-local ownership set, $pt \in 2^{\mathbb{A}}$ is the (ghost) set of local page table addresses, $mode \in \mathbb{B}$ is the translation mode (translated or untranslated) and $rls_l, rls_s, rls_{pt} \in 2^{\mathbb{A}}$ are the (ghost) release sets for local, shared and page table addresses respectively. For components X of thread local configuration $c.ts_{[i]}$ we abbreviate $c.X_{[i]}$. By $c.ghst_{[i]}$ we abbreviate the ghost information of thread i (except the dirty flag) and the shared ghost information:

$$c.ghst_{[i]} = (c.\mathcal{O}_{[i]}, c.pt_{[i]}, c.rls_{l[i]}, c.rls_{s[i]}, c.rls_{pt[i]}, c.shared, c.ro).$$

For the union of all release threads of thread i we abbreviate $c.rls_{[i]}$:

$$c.rls_{[i]} = c.rls_{l[i]} \cup c.rls_{s[i]} \cup c.rls_{pt[i]}.$$

Ownership transfer. Let $ghst = (\mathcal{O}, pt, rls_l, rls_s, rls_{pt}, shared, ro)$ be the ghost information of thread i. Then the ownership transfer performed by a ghost, volatile write or RMW instruction I in thread i is defined as:

$$otran(ghst, i, I) = (\mathcal{O}', pt', rls'_l, rls'_s, rls'_{pt}, shared', ro'),$$

where the ownership sets change according to Fig. 2 and the release sets accumulate released addresses if $G(I)$ and are cleared otherwise:

$$\begin{aligned}
ro' &= ro \cup (I.R \setminus I.W) \setminus (I.A \cup I.A_{pt}) & \mathcal{O}' &= \mathcal{O} \cup I.A \setminus I.R \\
shared' &= shared \cup I.R \cup I.R_{pt} \setminus (I.L \cup I.A_{pt}) & pt' &= pt' \cup I.A_{pt} \setminus I.R_{pt} \\
rls'_s &= G(I)\ ?\ rls_s \cup (I.R \cap shared) : \emptyset & rls'_{pt} &= G(I)\ ?\ rls_{pt} \cup I.R_{pt} : \emptyset \\
rls'_l &= G(I)\ ?\ rls_l \cup (I.R \setminus shared) : \emptyset.
\end{aligned}$$

Release set rls_s accumulates the shared fraction of released addresses (i.e., before the release these addresses were owned and shared), rls_l - the unshared fraction and rls_{pt} - released page table addresses.

Semantics. The computation of the virtual machine is defined by a non-deterministic transition relation $c \Longrightarrow c'$, where every step is either a program step, a memory step, an MMU step or a page fault step of thread i.

$$\frac{c \overset{\mathrm{p}}{\Longrightarrow}_i c' \vee c \overset{\mathrm{m}}{\Longrightarrow}_i c' \vee c \overset{\mathrm{mu}}{\Longrightarrow}_i c' \vee c \overset{\mathrm{pf}}{\Longrightarrow}_i c'}{c \Longrightarrow_i c'} \qquad \frac{c \Longrightarrow_i c'}{c \Longrightarrow c'}$$

A program step of thread i applies (uninterpreted) function δ_p to the program state and the set of temporaries of the thread to obtain a new program state and newly generated instructions, which are then appended to the old instruction list (Fig. 3). For a newly generated read or RMW instruction I we assume the read temporary $I.t$ to be fresh i.e., every read has to be done to a new temporary[4]. We formalize this assumption in Sect. 5.5.

A memory step of thread i is defined by a case split on the type of instruction $I = hd(c.is_{[i]})$ to be executed (Fig. 3). In case of a read, write or RMW instruction we first translate the virtual address $I.va$ using the current MMU state and chose a physical address pa from the set of available address translations provided by function $atran$. Hence, to execute such an instruction there has to be at least one possible address translation available. For a read instruction we update the value of temporary $I.t$ with the read value $c.m(pa)$. For a write instruction we obtain the write value by applying the function $I.f$ to the current set of temporaries and store the write value at memory address pa. In case of a volatile write we also perform the ownership transfer and set the dirty bit. An RMW instruction first performs a read of memory cell $c.m(pa)$ into temporary $I.t$ and then checks condition $I.cond$ on the updated set of temporaries. If the test succeeds we obtain the write value by applying $I.f$ to the updated set of temporaries, store this value at address pa and perform the ownership transfer. Independent on the test result we reset the dirty bit and clear the release sets. Fence and ghost instructions do not update the non-ghost part of the state (except reducing the length of the instruction list). For a ghost instruction we just perform the ownership transfer and for a fence instruction we clear the release sets and reset the dirty bit. Mode switch, INVLPG and write to PTO instructions also clear the release sets and reset the dirty bit as a side effect. In case of a mode switch we change the translation mode to $I.mode$. INVLPG instruction removes the invalidated translation from the MMU using function δ_{flush} and a write to PTO instruction applies function δ_{wpto} to the current MMU state.

MMU of thread i (Fig. 4) can either perform a read from the page tables updating the MMU state or a write setting control bits in the page tables. In case of a read the new MMU state is chosen from the set of MMU states provided by function δ_{mmur} and in case of a write we chose the value to be written to the memory from the set of values provided by function δ_{mmuw}. A page fault step is triggered when we are running in translated mode, in the head of the instruction list there is an instruction which requires address translation and

[4] When instantiation the model one can easily discharge this assumption by attaching a time stamp to every read destination.

$$\frac{(p', is') = \delta_p(c.p_{[i]}, c.\vartheta_{[i]})}{c \xrightarrow{\mathrm{p}}_i c[p_{[i]} := p', is_{[i]} := c.is_{[i]} \circ is']}$$

$$\frac{R(I) \quad pa \in (atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r))}{c \xrightarrow{\mathrm{m}}_i c[\vartheta_{[i]} := c.\vartheta_{[i]}(I.t \mapsto c.m(pa)), is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{nvW(I) \quad pa \in (atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r))}{c \xrightarrow{\mathrm{m}}_i c[m := c.m(pa \mapsto I.f(\vartheta_{[i]})), is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{vW(I) \quad pa \in (atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r)) \quad ghst' = otran(c.ghst_{[i]}, i, I)}{c \xrightarrow{\mathrm{m}}_i c[m := c.m(pa \mapsto I.f(\vartheta_{[i]})), ghst_{[i]} := ghst', \mathcal{D}_{[i]} := 1, is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{\begin{array}{c}RMW(I) \quad pa \in (atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r)) \quad \vartheta' = c.\vartheta_{[i]}(I.t \mapsto c.m(pa)) \\ m' = c.m(pa \mapsto I.f(\vartheta')) \quad I.cond(\vartheta') \quad ghst' = otran(c.ghst_{[i]}, i, I)\end{array}}{c \xrightarrow{\mathrm{m}}_i c[m := m', \vartheta_{[i]} := \vartheta', ghst_{[i]} := ghst', \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{\begin{array}{c}RMW(I) \quad pa \in (atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r)) \\ \vartheta' = c.\vartheta_{[i]}(I.t \mapsto c.m(pa)) \quad \neg I.cond(\vartheta')\end{array}}{c \xrightarrow{\mathrm{m}}_i c[\vartheta_{[i]} := \vartheta', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{G(I)}{c \xrightarrow{\mathrm{m}}_i c[ghst_{[i]} := otran(c.ghst_{[i]}, i, I), is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{FENCE(I)}{c \xrightarrow{\mathrm{m}}_i c[rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{SWITCH(I)}{c \xrightarrow{\mathrm{m}}_i c[mode_{[i]} := I.mode, rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{INVLPG(I) \quad mmu' = \delta_{flush}(c.mmu_{[i]}, I.F, I.fi)}{c \xrightarrow{\mathrm{m}}_i c[mmu_{[i]} := mmu', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]})]}$$

$$\frac{WPTO(I) \quad mmu' = \delta_{wpto}(c.mmu_{[i]}, I.v)}{c \xrightarrow{\mathrm{m}}_i c[mmu_{[i]} := mmu', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(c.is_{[i]})]}$$

Fig. 3: Program and memory steps of thread i of the virtual machine. For the memory step we assume $I = hd(c.is_{[i]})$.

$$\frac{mode_{[i]} \quad can\text{-}access(c.mmu_{[i]}, pa) \quad mmu' \in (\delta_{mmur}(c.mmu_{[i]}, pa, c.m(pa)))}{c \xRightarrow{\text{mu}}_i c[mmu_{[i]} := mmu']}$$

$$\frac{mode_{[i]} \quad can\text{-}access(c.mmu_{[i]}, pa) \quad v' \in (\delta_{mmuw}(c.mmu_{[i]}, pa, c.m(pa)))}{c \xRightarrow{\text{mu}}_i c[m := c.m(pa \mapsto v')]}$$

$$\frac{\begin{array}{c} mode_{[i]} \quad can\text{-}access(c.mmu_{[i]}, pa) \quad I = hd(c.is_{[i]}) \quad (R(I) \vee W(I) \vee RMW(I)) \\ can\text{-}page\text{-}fault(c.mmu_{[i]}, I.va, I.r, pa, c.m(pa)) \\ p' = \delta_{pf}(c.p_{[i]}, I.va, I.r, c.m(pa)) \quad mmu' = \delta_{flush}(c.mmu_{[i]}, \{I.va\}, 0) \end{array}}{c \xRightarrow{\text{pf}}_i c[is_{[i]} := [], p_{[i]} := p', mmu_{[i]} := mmu', \mathcal{D}_{[i]} := 0, rls_{[i]} := \emptyset]}.$$

Fig. 4: MMU steps and page fault step of thread i of the virtual machine.

the page fault for the address of the instruction can be signalled. As an effect of the page fault we (i) update the program state using (uninterpreted) function δ_{pf} which loads the information about the faulty translation to the program state, (ii) flush all translations for the faulty virtual address from the MMU and (iii) clear the instruction list. As a side effect we also empty the release sets and reset the dirty bit.

Note, that reading a faulty entry and signalling a page fault is done in a single atomic transition, i.e. the MMU is not allowed to pre-fetch a faulty PTE first and then use it for signalling a page fault some time later. This allows to model silent rights granting in page tables i.e., when the user grants more rights in a PTE without a consequent TLB flush, and setting of present bit in a PTE without TLB flushing. In a real TLB of the x86 machine the same behaviour can be achieved by performing a fresh re-walk of page tables in case of a page fault [1]

By $c \Rightarrow^k c'$ we denote that state c' is reachable from c in exactly k step and by $c \Rightarrow^* c'$ we denote the reflexive transitive closure of $\Rightarrow$. We also use the same kind of notation when arguing about executions of thread i and executions which consist only of certain kinds of steps (e.g., only memory steps).

Safety. Safety for instruction I in thread i (Fig. 5) restricts the sets of translated physical addresses which can be accessed by read, write and RMW instructions and sets the rules for the ownership transfer. A translated physical address of the volatile read instruction can be either owned by the thread, or shared, or can belong to local page tables. Moreover, we have to make sure that the dirty bit is cleared before we can execute a volatile read. A non-volatile read can only be performed to an owned or read-only address. In case of a volatile write we require the target address to be not present in the ownership and page table sets of other threads and to be excluded from the read only addresses. A non-volatile write can only be performed to owned unshared addresses. For RMW instruc-

$$
\begin{aligned}
&safe\text{-}instr(c,i,I) \equiv (\forall pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).\\
&\quad (vR(I) \to pa \in c.\mathcal{O}_{[i]} \cup c.shared \cup c.pt_{[i]} \wedge \neg c.\mathcal{D}_{[i]}) \wedge\\
&\quad (nvR(I) \to pa \in c.\mathcal{O}_{[i]} \cup c.ro) \wedge\\
&\quad (vW(I) \to \forall j \neq i.\ pa \notin c.\mathcal{O}_{[j]} \cup c.pt_{[j]} \wedge pa \notin c.ro) \wedge\\
&\quad (nvW(I) \to pa \in c.\mathcal{O}_{[i]} \wedge pa \notin c.shared) \wedge\\
&\quad (RMW(I) \wedge \neg I.cond(\vartheta') \to pa \in c.\mathcal{O}_{[i]} \cup c.shared \cup c.pt_{[i]}) \wedge\\
&\quad (RMW(I) \wedge I.cond(\vartheta') \to \forall j \neq i.\ pa \notin c.\mathcal{O}_{[j]} \cup c.pt_{[j]} \wedge pa \notin c.ro)) \wedge\\
&\quad (vW(I) \vee G(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \to \forall j \neq i.\ I.L \subseteq I.A \wedge\\
&\qquad (I.A \cup I.A_{pt}) \cap (c.\mathcal{O}_{[j]} \cup c.pt_{[j]}) = \emptyset \wedge I.R \subseteq c.\mathcal{O}_{[i]} \wedge I.R_{pt} \subseteq c.pt_{[i]} \wedge\\
&\qquad I.A \subseteq c.\mathcal{O}_{[i]} \cup c.shared \cup I.R_{pt} \wedge I.A_{pt} \subseteq c.pt_{[i]} \cup c.shared \cup I.R \wedge\\
&\qquad I.A \cap I.R = \emptyset \wedge I.A_{pt} \cap I.R_{pt} = \emptyset \wedge I.A_{pt} \cap I.A = \emptyset)
\end{aligned}
$$

Fig. 5: Safety of instruction I in thread i, where $\vartheta' = c.\vartheta_{[i]}(I.t \mapsto c.m(pa))$.

tions we split cases depending on the result of the RMW test. We treat RMW as a volatile read if the test fails and as a volatile write if the test succeeds. For instructions performing the ownership transfer we require (i) the local fraction $I.L$ of acquired addresses to be a subset of the acquired addresses $I.A$, (ii) acquired addresses $I.A \cup I.A_{pt}$ to be disjoint with the ownership and page table sets of other threads, (iii) released addresses $I.R$ to be a subset of the addresses owned by the thread and released page table addresses $I.R_{pt}$ to be a subset of the local page table addresses, (iv) acquired addresses $I.A$ to be a subset of owned, shared and released page table addresses, (v) acquired page table addresses $I.A_{pt}$ to be a subset of page table, shared and released addresses, (vi) acquired addresses $I.A$ must be disjoint with released addresses $I.R$ and acquired page table addresses must be disjoint with released addresses $I.R_{pt}$ and (vii) acquired addresses $I.A$ must be disjoint with the acquired page table addresses $I.A_{pt}$.

An MMU step reading or writing physical address pa is safe if pa belongs a local page table or to the shared portion of the memory which is not owned by anyone and which does not belong to the read only memory:

$$safe\text{-}mmu\text{-}acc(c,pa,i) \equiv pa \in c.pt_{[i]} \cup c.shared \wedge pa \notin c.ro \wedge \forall j.\ pa \notin c.\mathcal{O}_{[j]}.$$

Configuration c of the virtual machine is safe if first instructions in the instruction lists of all threads are safe and all MMU steps, which can be performed from c are safe:

$$
\begin{aligned}
safe\text{-}state(c) \equiv{}& \forall i.\ safe\text{-}instr(c,i,hd(c.is_{[i]})) \wedge\\
&\forall i, pa.\ can\text{-}access(c.mmu_{[i]}, pa) \to safe\text{-}mmu\text{-}acc(c,pa,i).
\end{aligned}
$$

Predicate $safe\text{-}reach(c,n)$ denotes that any configuration reachable from configuration c in at most n steps is safe: If we omit the number of steps, then

the predicate denotes that any configuration reachable from c is safe :

$$safe\text{-}reach(c, n) \equiv safe\text{-}state(c) \wedge \forall c'.\ \forall k \leq n.\ c \Rightarrow^{k} c' \rightarrow safe\text{-}state(c')$$
$$safe\text{-}reach(c) \equiv \forall c'.\ c \Rightarrow^{*} c' \rightarrow safe\text{-}state(c).$$

When execution of a virtual machine starts from the initial state and is safe, we can be sure that certain relations between various ownership sets are maintained. We gather these properties in the following predicate:

$$\begin{aligned} disjoint\text{-}osets(c) \equiv\ & c.ro \subseteq c.shared \wedge \forall i.\ \forall j \neq i. \\ & c.\mathcal{O}_{[i]} \cap c.\mathcal{O}_{[j]} = \emptyset \wedge c.\mathcal{O}_{[i]} \cap c.ro = \emptyset \wedge \\ & c.\mathcal{O}_{[i]} \cap c.pt_{[j]} = \emptyset \wedge c.pt_{[i]} \cap c.pt_{[j]} = \emptyset \wedge \\ & c.\mathcal{O}_{[i]} \cap c.pt_{[i]} = \emptyset \wedge c.pt_{[i]} \cap c.shared = \emptyset. \end{aligned}$$

Initial Configuration. The initial configuration of virtual machine is defined as

$$initial(c) \equiv disjoint\text{-}osets(c) \wedge \forall i. c.rls_{[i]} = \emptyset \wedge c.is_{[i]} = [].$$

4.4 Store Buffer Machine

Our SB machine contains all the components from the virtual machine plus thread-local SBs. The ghost fields carried from the virtual machine configuration are used to simplify the proof, particularly they allow to specify properties of the stores present in the SB without referring to the corresponding configuration of the virtual machine. Store buffers are used not only to buffer memory stores, but also to collect history information about the executed memory and program steps. The ghost fields carried from the virtual machine do not influence the execution of the SB machine in any way. The history information which is recorded in the SB also does not have any influence on the non-ghost components (except of the length of the SB when the history information retires). Hence, proving simulation between an SB machine without the ghost and history components and with them is a trivial task and we omit it here.

Configuration. Configuration of the SB machine c_{sbh} has the same components as configurations of the virtual machine:

$$c_{sbh} = (m, shared, ro, ts) \in \mathbb{M}_{sbh}.$$

Thread-local configuration $c_{sbh}.ts_{[i]}$ has all components from the local configuration of the virtual machine plus an SB component:

$$c_{sbh}.ts_{[i]} = (p, is, \vartheta, mmu, pt, mode, \mathcal{D}, \mathcal{O}, rls_l, rls_s, rls_{pt}, sb) \in \mathbb{K}_{sbh},$$

where $sb \in \mathbb{I}_{sb}^{*}$ is a sequence of *SB instructions*:

$$\begin{aligned} \mathbb{I}_{sb} = \ & \mathbf{Write_{sb}}\ \text{vol va (D, f) r annot pa v} \mid \mathbf{Read_{sb}}\ \text{vol va t r pa v} \\ & \mid \mathbf{Ghost_{sb}}\ \text{annot} \mid \mathbf{Prog_{sb}}\ \text{p p}'\ \text{is}. \end{aligned}$$

The only SB instruction with a non-ghost effect is $\mathbf{Write_{sb}}$, which stores value v to memory address pa when it leaves the SB. The other fields of $\mathbf{Write_{sb}}$ collect the history information and are carried over from the corresponding **Write** instruction, when it is executed and is put to the SB. When read, ghost instructions or program steps are executed we also record the ghost information for them in the SB. In case of a read we additionally record physical address pa from where the read was performed and value v that was read. For program steps we record program state p of the thread configuration before the step and program state p' after the step together with the newly generated instruction sequence is.

We overload predicates $R(I)$, $W(I)$, etc., to work also on SB instructions and introduce predicate $P(I)$ for the recorded program step.

Partial function $sbins : \mathbb{I} \times \mathbb{A} \times \mathbb{V} \rightharpoonup \mathbb{I}_{sb}$ converts a read, write or ghost memory instruction to a corresponding SB instruction:

$$sbins(I, pa, v) = \begin{cases} \mathbf{Read_{sb}} \text{ vol va t r pa v} & I = \mathbf{Read} \text{ vol va t r} \\ \mathbf{Write_{sb}} \text{ vol va (D, f) r annot pa v} & I = \mathbf{Write} \text{ vol va (D, f) r annot} \\ \mathbf{Ghost_{sb}} \text{ annot} & I = \mathbf{Ghost} \text{ annot.} \end{cases}$$

The function $ins \in \mathbb{I}_{sb} \rightharpoonup \mathbb{I}$ performs conversion in the other direction:

$$ins(I) = \begin{cases} \mathbf{Read} \text{ vol va t r} & I = \mathbf{Read_{sb}} \text{ vol va t r pa v} \\ \mathbf{Write} \text{ vol va (D, f) r annot} & I = \mathbf{Write_{sb}} \text{ vol va (D, f) r annot pa v} \\ \mathbf{Ghost} \text{ annot} & I = \mathbf{Ghost_{sb}} \text{ annot.} \end{cases}$$

We also define an overloaded version of the function ins which operates on a list of SB instructions:

$$ins(sb) = \begin{cases} [] & sb = [] \\ ins(tl(sb)) & P(hd(sb)) \\ ins(hd(sb)) \circ ins(tl(sb)) & \text{otherwise.} \end{cases}$$

For components X of thread local configuration $c_{sbh}.ts_{[i]}$ we simply write $X_{[i]}$ if configuration c_{sbh} is clear from the context. Note, that we completely omit the configuration identifier **only** for the SB machine, and always write it for the virtual machines in order to avoid confusion. As in the case of the virtual machine, we abbreviate by $rls_{[i]}$ the union of all release sets of thread i and by $ghst_{[i]}$ we denote the ghost information of thread i (excluding the dirty flag) and the shared ghost information.

The history information for reads, program steps and ghost operations allows to keep track of instructions which have been executed in the store buffer machine after preceding write instruction is executed, but before it leaves the SB. We use this information in Sect. 5.2 to couple the state of SB and virtual machines in

the simulation theorem. The history information for writes keeps track of the store values and the physical address chosen for the address translation. This information is coupled with the current state of the SB machine with the help of additional invariants (Sect. 5.4). These invariants together with the coupling relation guarantee, for instance, that we can chose the same address translation when executing the corresponding instruction in the virtual machine and that the store value of that instruction in the virtual machine will be the same as in the SB machine.

Semantics. The computation of the SB machine is defined by a non-deterministic transition relation $c_{sbh} \Rightarrow c'_{sbh}$, where every step is either a program step, a memory step, an SB step, an MMU step or a page fault step of thread i:

$$\frac{c_{sbh} \overset{\text{p}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{m}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{mu}}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{\text{pf}}{\Longrightarrow}_i c'_{sbh}}{c_{sbh} \Longrightarrow_i c'_{sbh}}$$

A program step of the SB machine has the same effect as in the virtual machine and is recorded as history information in the SB:

$$\frac{(p', is') = \delta_p(p_{[i]}, \vartheta_{[i]}) \quad I = PROG_{sb}\ p_{[i]}\ p'\ is'}{c_{sbh} \overset{\text{p}}{\Longrightarrow}_i c_{sbh}[p_{[i]} := p', sb_{[i]} := sb_{[i]} \circ I, is_{[i]} := is_{[i]} \circ is']}$$

MMU read and write steps of the SB machine have the same semantics as in the virtual machine and we don't state them here. The page fault step can occur only when the SB is empty:

$$\frac{\begin{array}{c} mode_{[i]} \quad can\text{-}access(mmu_{[i]}, pa) \quad I = hd(is_{[i]}) \quad (R(I) \vee W(I) \vee RMW(I)) \\ can\text{-}page\text{-}fault(mmu_{[i]}, I.va, I.r, pa, m(pa)) \quad sb_{[i]} = [] \\ p' = \delta_{pf}(p_{[i]}, I.va, I.r, m(pa)) \quad mmu' = \delta_{flush}(mmu_{[i]}, \{I.va\}, 0) \end{array}}{c_{sbh} \overset{\text{pf}}{\Longrightarrow}_i c_{sbh}[is_{[i]} := [], p_{[i]} := p', mmu_{[i]} := mmu', rls_{[i]} := \emptyset]}$$

A read instruction performs the read and is recorded to the SB as history information (Fig. 6). The read value is obtained with the function $fwd(sb_{[i]}, m, pa)$, which forwards the first store value to pa which is present in the SB or returns the memory value $m(pa)$ if there are no writes to pa in the SB:

$$fwd(sb, m, a) = \begin{cases} m(a) & sb = [] \\ I.v & I = last(sb) \wedge W(I) \wedge I.pa = a \\ fwd(sb[0 : |sb| - 2], m, a) & \text{otherwise.} \end{cases}$$

A write instruction is not executed immediately, but is buffered in the SB together with the ghost history information. A ghost instruction is also recorded in the SB without an immediate effect on the configuration. All the other memory instructions can be executed only when SB is empty and have the same semantics as defined for the virtual machine. Note, that SB collects read, write

$$\frac{R(I) \quad pa \in (atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \quad v = fwd(sb_{[i]}, m, pa)}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[\vartheta_{[i]} := \vartheta_{[i]}(I.t \mapsto v), sb_{[i]} := sb_{[i]} \circ [sbins(I, pa, v)], is_{[i]} := tl(is_{[i]})]}$$

$$\frac{W(I) \quad pa \in (atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \quad \mathcal{D}' = vW(I) \vee \mathcal{D}_{[i]}}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[sb_{[i]} := sb_{[i]} \circ [sbins(I, pa, I.f(\vartheta_{[i]}))], is_{[i]} := tl(is_{[i]}), \mathcal{D}_{[i]} := \mathcal{D}']}$$

$$\frac{\begin{array}{c}RMW(I) \quad pa \in (atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \quad \vartheta' = \vartheta_{[i]}(I.t \mapsto m(pa)) \\ sb_{[i]} = [] \quad m' = m(pa \mapsto I.f(\vartheta')) \quad I.cond(\vartheta') \quad ghst' = otran(ghst_{[i]}, i, I)\end{array}}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[m := m', \vartheta_{[i]} := \vartheta', ghst_{[i]} := ghst', \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{\begin{array}{c}RMW(I) \quad pa \in (atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \\ sb_{[i]} = [] \quad \vartheta' = \vartheta_{[i]}(I.t \mapsto m(pa)) \quad \neg I.cond(\vartheta')\end{array}}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[\vartheta_{[i]} := \vartheta', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{G(I)}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[sb_{[i]} := sb_{[i]} \circ [sbins(I, pa, v)], is_{[i]} := tl(is_{[i]})]}$$

$$\frac{FENCE(I) \quad sb_{[i]} = []}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{SWITCH(I) \quad sb_{[i]} = []}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[mode_{[i]} := I.mode, rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{INVLPG(I) \quad sb_{[i]} = [] \quad mmu' = \delta_{flush}(mmu_{[i]}, I.F, I.fi)}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[mmu_{[i]} := mmu', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

$$\frac{WPTO(I) \quad sb_{[i]} = [] \quad mmu' = \delta_{wpto}(mmu_{[i]}, I.v)}{c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c_{sbh}[mmu_{[i]} := mmu', rls_{[i]} := \emptyset, \mathcal{D}_{[i]} := 0, is_{[i]} := tl(is_{[i]})]}$$

Fig. 6: Memory step of thread i of the SB machine, where $I = hd(is_{[i]})$.

$$\frac{W(I) \quad ghst' = (nvW(I)\ ?\ ghst_{[i]} : otran(ghst_{[i]}, i, I))}{c_{sbh} \xRightarrow{\text{sb}}_i c_{sbh}[m := m(I.pa \mapsto I.v), ghst_{[i]} := ghst', sb_{[i]} := tl(sb_{[i]})]}$$

$$\frac{R(I) \vee P(I)}{c_{sbh} \xRightarrow{\text{sb}}_i c_{sbh}[sb_{[i]} := tl(sb_{[i]})]} \qquad \frac{G(I) \quad ghst' = otran(ghst_{[i]}, i, I)}{c_{sbh} \xRightarrow{\text{sb}}_i c_{sbh}[ghst_{[i]} := ghst', sb_{[i]} := tl(sb_{[i]})]}$$

Fig. 7: Store buffer step of thread i of the SB machine, where $I = hd(sb_{[i]})$.

and ghost instructions annotated with additional (ghost) history information. Among those instructions only writes contain non-ghost data and perform an update of the non-ghost part of the configuration when they leave the SB.

An SB step of thread i is defined in Fig. 7. When a write instruction leaves the SB, then it deliverers a buffered store to the memory and performs an ownership transfer if the write is volatile. A ghost instruction only performs an ownership transfer. Read and program instructions are simply skipped.

5 Store Buffer Reduction

5.1 Reduction Theorem

Our main result is a simulation theorem between the SB machine and the virtual machine.

Theorem 1 (Reduction).

$$c_{sbh} \Rightarrow^* c'_{sbh} \wedge c_{sbh} \sim c \wedge initial(c) \wedge sbempty(c_{sbh}) \wedge safe\text{-}reach(c) \rightarrow$$
$$\exists c'.\ c \Rightarrow^* c' \wedge c'_{sbh} \sim c'$$

We consider only executions which start with empty SBs:

$$sbempty(c_{sbh}) \equiv \forall i.\ c_{sbh}.sb_{[i]} = [].$$

Initial configuration of the SB machine can be obtained from the initial configuration of the virtual machine by simply copying all components. The coupling relation $c_{sbh} \sim c$ should (at least) guarantee equality of the local thread configurations (with the exception of the SB component) when the SBs are empty (we define it formally in Sect. 5.2).

We do the proof of Theorem 1 on step by step basis i.e., for every step of the SB machine we find a (possibly empty) corresponding sequence of steps of the virtual machine in such a way, that the coupling relation is maintained. The main property we have to prove in order to maintain the coupling relation, particularly to make sure that local configurations are consistent when the SBs are empty, is that the reads (including the MMU reads) performed in both

machines get the same value. As a result, the crucial role plays the scheduling of the virtual machine. The most straightforward approaches one could think about are (i) executing an instruction on the virtual machine when this instruction is executed on the SB machine and (ii) executing an instruction on the virtual machine when this instruction leaves the SB (i.e., delaying the virtual machine until this point). The history information recorded in the SB in this case helps to reconstruct the instructions which yet have to be executed in the virtual machine. However, both of these approaches do not work. In the first case we get a problem when thread i executes a volatile write and puts it to the store buffer and then thread j executes a volatile read. In the virtual machine the result of the write will already be committed to the memory and thread j will read the new value, while in the SB machine thread j will get the old value, because the write is still present in the store buffer of thread i. The same example also rules out the second approach: if we delay a volatile read of thread j in the virtual machine, then it might be scheduled after the volatile write of thread i leaves the SB, and the virtual machine will again read the new value.

Hence, to guarantee the consistency of read results in both executions we have to schedule the virtual machine in such a way, that

- a volatile write must be delayed in the virtual machine until the volatile write exits the SB in the SB machine,
- a volatile read must be executed simultaneously in both machines. Our programming discipline guarantees that when a volatile read is executed there can be no volatile writes in the SB of the SB machine.

As a result, the shared portion of the memory will be always consistent between the machines. The page tables in that sense are also considered as part of the "shared" memory, even if these page tables are thread-local. Indeed, if the content of local page tables would be inconsistent between the machines, then MMU reads in the virtual machine would either read different values (due to the absence of SB forwarding for MMU reads) or would have to be delayed until the competing volatile writes to local page tables leave the SB. However delaying MMU reads in the virtual machine is also not feasible, because that would force us to delay subsequent MMU writes. These MMU writes might be performed to shared page tables (we do allow a local PTE to point to a shared PTE), which would lead to inconsistent shared memory. As a result, we have to execute all MMU steps simultaneously in both machines. Together with the possible delay in instruction execution this leads to reordering of MMU steps with respect to executed instructions in a given thread, but this reordering is always done to the left of the instruction sequence (Fig. 8). This behaviour is fine, because the monotonicity property of our MMU model guarantees that once added the address translations are never removed from the MMU. In the virtual machine some address translations will be added to the MMU earlier than in the SB machine (if one counts time by the number of executed instructions), but they will still remain there when the instructions which might rely on these address translations are executed.

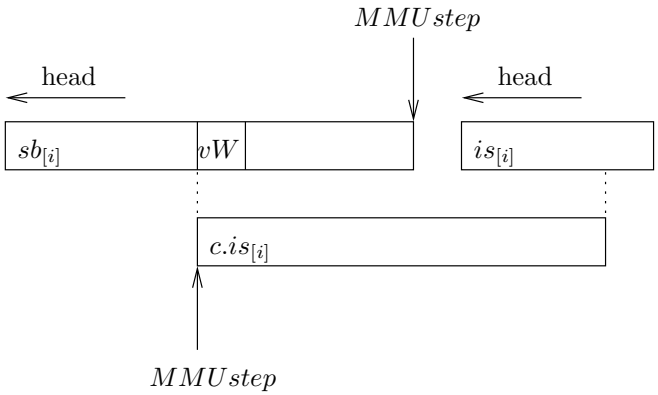

Fig. 8: Reordering of MMU steps.

Note, that the scheduling for instructions performing local memory accesses is not so crucial, because our programming discipline guarantees that these accesses never race with memory accesses of other threads and with memory accesses performed by MMUs, including the MMU of the executing thread itself. By a race here we understand two competing accesses where at least one of them is a write.

The following scheduling policy satisfies all the conditions stated above:

- when a volatile write is executed in the SB machine, the virtual machine is delayed and does not make any steps,
- when a volatile read is executed in the SB machine, the virtual machine executes the same step,
- when a non-volatile memory access, a ghost instruction or a program step of thread i is executed in the SB machine we make a case split on whether the SB of thread i contains a volatile write or not. In case it does, then execution of thread i in the virtual machine is already delayed (it is waiting until the volatile write will leave the SB) and we do not make any steps. In case it doesn't, then the virtual machine executes the same step of thread i,
- all the other instructions and the page fault step require the SB to be empty before they can be executed. Hence, we execute these steps simultaneously in both machines,
- when a volatile write exits the SB, the virtual machine executes this volatile write and all instructions and program steps recorded in the SB until the next volatile write (or until the end of the SB, if there are no other volatile writes there),
- when a read, non-volatile write, a ghost instruction or the recorded program step exits the SB, the virtual machine does not perform any steps, because it has already performed the corresponding step before,
- MMU steps are always executed simultaneously in both machines.

As a result of the rules stated above, the virtual machine is on-par or behind the SB machine in terms of executed memory steps (instructions) and program steps and it is always on-par with the SB machine in terms of executed MMU steps. However, in terms of the stores committed to the memory the virtual machine is either on-par or ahead of the SB machine. In the next section we define the coupling relation $c_{sbh} \sim c$ which captures the essence of our scheduling policy.

5.2 Coupling Relation

The part of the SB after and including the first volatile write is called *suspended*, because these steps are not yet executed on the virtual machine. The part of the SB before the first volatile write (or the whole SB if it does not have any volatile writes) is called *executed*, since the virtual machine has already performed these steps. We introduce the functions $exec(sb)$ and $susp(sb)$, which return the executed and the suspended parts of the SB respectively:

$$exec(sb) = \begin{cases} sb[0:k-1] & k = min\{j \mid vW(sb[j])\} \\ sb & \text{no vW in } sb. \end{cases}$$

$$susp(sb) = \begin{cases} sb[k:|sb|-1] & k = min\{j \mid vW(sb[j])\} \\ [] & \text{no vW in } sb. \end{cases}$$

In contrast to a non-deterministic memory transition an SB step is always deterministic. To simplify the notation we introduce a function δ_{sb}, which computes the next state of the SB machine after an SB step of thread i or returns the unmodified machine state in case the SB of thread i is empty:

$$\delta_{sb}(c_{sbh}, i) \equiv \begin{cases} c'_{sbh} & sb_{[i]} \neq [] \wedge c_{sbh} \overset{\text{sb}}{\Longrightarrow}_i c'_{sbh} \\ c_{sbh} & sb_{[i]} = []. \end{cases}$$

Configuration of the machine after executing k steps of the SB of thread i is defined inductively as

$$\delta^k_{sb}(c_{sbh}, i) = \begin{cases} c_{sbh} & k = 0 \\ \delta_{sb}(\delta^{k-1}_{sb}(c_{sbh}, i), i) & \text{otherwise.} \end{cases}$$

Function $\Delta_{sb}(c_{sbh}, i)$ executes all instruction in the SB of thread i and function $\Delta^{exec}_{sb}(c_{sbh}, i)$ executes all instructions before the first volatile write:

$$\Delta_{sb}(c_{sbh}, i) = \delta^{|sb_{[i]}|}_{sb}(c_{sbh}, i)$$
$$\Delta^{exex}_{sb}(c_{sbh}, i) = \delta^{|exec(sb_{[i]})|}_{sb}(c_{sbh}, i).$$

We overload functions $\Delta_{sb}(c_{sbh})$ and $\Delta^{exec}_{sb}(c_{sbh})$ (leaving out the thread id) to compute the machine configuration after consecutive execution of instructions

$$
\begin{aligned}
&\forall X \in \{shared, ro, m\}.\ c.X = \Delta_{sb}^{exec}(c_{sbh}).X \wedge \\
&\forall i.\ \forall X \in \{\mathcal{O}, pt, rls_l, rls_s, rls_{pt}\}.\ c.X_{[i]} = \Delta_{sb}^{exec}(c_{sbh}, i).X_{[i]} \wedge \\
&\quad c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]})) = ins(susp(sb_{[i]})) \circ is_{[i]} \wedge \\
&\quad c.\vartheta_{[i]} = del\text{-}t(\vartheta_{[i]}, susp(sb_{[i]})) \wedge c.p_{[i]} = hd\text{-}p(p_{[i]}, susp(sb_{[i]})) \wedge \\
&\quad c.mode_{[i]} = mode_{[i]} \wedge c.mmu_{[i]} = mmu_{[i]} \wedge \\
&\quad (c.\mathcal{D}_{[i]} \vee \exists I \in sb_{[i]}.\ vW(I) \leftrightarrow \mathcal{D}_{[i]}).
\end{aligned}
$$

Fig. 9: Definition of the coupling relation $c_{sbh} \sim c$.

from SBs of all threads, starting with thread id 0:

$$
\begin{aligned}
\Delta_{sb}(c_{sbh}) &= \Delta_{sb}(...\Delta_{sb}(\Delta_{sb}(c_{sbh}, 0), 1)..., np-1) \\
\Delta_{sb}^{exec}(c_{sbh}) &= \Delta_{sb}^{exec}(...\Delta_{sb}^{exec}(\Delta_{sb}^{exec}(c_{sbh}, 0), 1)..., np-1).
\end{aligned}
$$

We define $\Delta_{sb[\neq i]}(c_{sbh})$ and $\Delta_{sb[\neq i]}^{exec}(c_{sbh})$ to do the same computation but excluding steps of thread i.

With this notation we can now define the coupling relation $c_{sbh} \sim c$ (Fig. 9).

- To get shared component $X \in \{shared, ro, m\}$ of the virtual machine we take the corresponding component of the SB machine and execute all instructions from the executed portions of SBs of all threads. Note, that since the executed parts of SBs do not contain volatile writes, the content of the shared memory is always consistent between two machines,
- For thread-local components $X \in \{\mathcal{O}, pt, rls_l, rls_s, rls_{pt}\}$ of thread i we take the corresponding component of the SB machine and execute all instructions from the executed portion of the SB of thread i.
- To couple the instruction list (Fig. 10) we first observe, that the instruction list in the virtual machine should contain all instructions from the suspended part of the SB (with the exception of the history information for program steps) plus the instructions from the instruction list of the SB machine. Note however, that some of the instructions in the SB machine might be generated by the program steps, which are suspended in the virtual machine. Instead of removing these instructions from the instruction list of the SB machine, in the coupling relation we append them to the instruction list of the virtual machine. The function $ins(susp(sb_{[i]}))$ removes the program steps from the suspended portion of the SB and converts the instructions recorded in the store buffer into regular memory instructions by throwing aways the additional history information. The function $p\text{-}ins(susp(sb_{[i]}))$ extracts instructions generated by the program steps recorded in the suspended portion of the SB:

$$
p\text{-}ins(sb) = \begin{cases} [] & sb = [] \\ is \circ p\text{-}ins(tl(sb)) & hd(sb) = \mathbf{Prog_{sb}}\ \mathrm{p}\ \mathrm{p}'\ \mathrm{is} \\ p\text{-}ins(tl(sb)) & \text{otherwise.} \end{cases}
$$

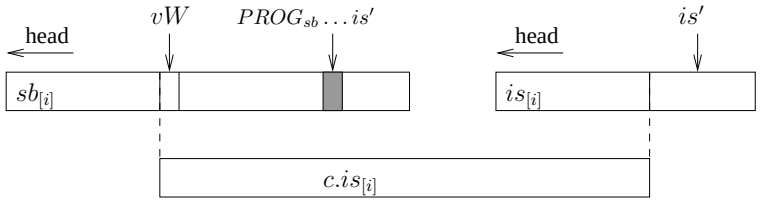

Fig. 10: Instruction list coupling.

– The set of temporaries of thread i of the virtual machine is obtained by removing all the temporaries used for reads in the suspended part of the SB, done by the function $del\text{-}t(\vartheta_{[i]}, sb)$:

$$del\text{-}t(\vartheta, sb) \equiv \vartheta \restriction_{dom(\vartheta) \setminus load_t(sb)} .$$

Since we assume all temporaries in the newly generated instructions to be fresh, we can be sure that we don't remove the temporaries which have been used for already executed reads.

– The program state in the virtual machine is obtained by function

$$hd\text{-}p(p_{[i]}, susp(sb_{[i]})),$$

which takes the recorded pre-state of the first program instruction in the suspended portion of the SB or simply takes the current program state in the SB machine if there are no suspended program instructions:

$$hd\text{-}p(p, sb) \equiv \begin{cases} p & sb = [] \\ p_1 & hd(sb) = \mathbf{Prog_{sb}}\ \mathrm{p}_1\ \mathrm{p}_2\ \mathrm{is} \\ hd\text{-}p(p, tl(sb)) & \text{otherwise.} \end{cases}$$

– The translation mode and the MMU state are always equal between the machines.
– The dirty bit in the SB machine is set iff it is also set in the virtual machine or if there is a volatile write in the (suspended part of the) SB.

Note, that the coupling relation defined here gives the full consistency between all components only when all the SBs are empty. As a result, one has to require the execution to end with a configuration where SBs are empty in order to use Theorem 1 to transfer all the results of the execution of the virtual machine to the SB machine. However, intermediate configurations, for instance those where only SBs of some threads are empty, can be also used to transfer partial execution results (e.g., for the memory content owned by a thread).

5.3 Safety of the Delayed Release

Our programming discipline essentially only allows races between volatile accesses of different threads and between MMU accesses. Practically, this means

that (i) while the reads are present in the suspended portion of the SB, the read results can not be invalidated by other threads and by MMUs and (ii) when a volatile read or an MMU read is executed in the SB machine, there can be no (non-volatile) writes to the same address in the executed portions of other threads. In the proof this for instance manifests in the following proof obligation: when a volatile write to pa leaves the SB of thread i, there are no (non-volatile) reads to pa in the suspended portions of SBs of other threads. We prove this by contradiction, assuming that such a read exists in the SB of thread j. In the corresponding configuration of the virtual machine this read is not yet executed. Hence, we forward thread j in the virtual machine configuration until the point where this read is at the head of the instruction list. From safety of all reachable traces of the virtual machine, we know that the resulting state is safe. Moreover, we can prove that for all reachable safe states of the virtual machine disjointness of the ownership sets is preserved. This implies a contradiction, because the safety of thread j requires pa to be either owned or read-only and the safety of thread i requires pa to be not owned by other threads and not read-only.

However, for some races the strategy described above does not work. Consider a case when thread i starting with an empty SB performs a non-volatile write to pa and then a ghost release of pa. Since the SB of thread i does not contain any volatile writes, these steps are immediately executed in the virtual machine. After that, MMU of thread i performs a read from pa. In the current trace of the virtual machine this operation is safe, since the address pa is not owned by any thread at the time of the MMU step. However, the read results in two machines will be inconsistent, because in the virtual machine the store to pa is already committed to the memory and in the virtual machine it is still present in the SB of thread i. To rule out this situation, we have to construct another unsafe trace of the virtual machine, which deviated from the current trace somewhere in the past. For the given example this means that we have to consider a trace where the MMU step is performed before the ghost release takes place. Construction of these deviated traces is not feasible in the step-by-step proof of Theorem 1, because there we only have safety of reachable traces starting from the current state of the virtual machine. To solve this problem we observe that the complications arise only when the addresses are released by ghost instructions. Information about these releases is collected into the (ghost) release sets. We use these sets to define *safety of the delayed release*, which can be used to rule out the described situation.

For a given instruction and for an MMU access safety of the delayed release is defined in Fig. 11. Safety of the delayed release for a given configuration and for reachable configurations is stated as

$$
\begin{aligned}
\mathit{safe\text{-}state}_d(c) &\equiv \forall i.\ \mathit{safe\text{-}instr}_d(c, i, \mathit{hd}(c.\mathit{is}_{[i]})) \wedge \\
&\qquad \forall i, a.\ \mathit{can\text{-}access}(c.\mathit{mmu}_{[i]}, \mathit{pa}) \rightarrow \mathit{safe\text{-}mmu\text{-}acc}_d(c, \mathit{pa}, i) \\
\mathit{safe\text{-}reach}_d(c, n) &\equiv \mathit{safe\text{-}state}_d(c) \wedge \forall c'.\ \forall k \leq n.\ c \Rightarrow^k c' \rightarrow \mathit{safe\text{-}state}_d(c') \\
\mathit{safe\text{-}reach}_d(c) &\equiv \forall n.\ \mathit{safe\text{-}reach}_d(c, n).
\end{aligned}
$$

$$
\begin{aligned}
&safe\text{-}instr_d(c,i,I) \equiv safe\text{-}instr(c,i,I) \wedge \\
&\quad \forall pa.\ \forall j \neq i.\ pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}) \rightarrow \\
&\quad (vR(I) \vee (RMW(I) \wedge \neg I.cond(\vartheta')) \rightarrow pa \notin c.rls_{l[j]} \cup c.rls_{pt[j]}) \wedge \\
&\quad (nvR(I) \vee vW(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \rightarrow pa \notin c.rls_{[j]}) \wedge \\
&\quad (vW(I) \vee G(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \rightarrow (I.A \cup I.A_{pt}) \cap c.rls_{[j]} = \emptyset) \\
&safe\text{-}mmu\text{-}acc_d(c,a,i) \equiv safe\text{-}mmu\text{-}acc(c,a,i) \wedge a \notin c.rls_{l[i]} \wedge \forall j \neq i.\ a \notin c.rls_{[j]}.
\end{aligned}
$$

Fig. 11: Safety of the delayed release for an instruction I in thread i and for an MMU access to address pa in thread i, where $\vartheta' = c.\vartheta_{[i]}(I.t \mapsto c.m(pa))$.

5.4 Invariants

In this section we define invariants $inv(c_{sbh})$ on the SB machine, which we later use in the simulation proof. We start with giving some auxiliary definitions.

The set of all addresses acquired (resp. released) by instructions in store buffer sb is defined as

$$
\begin{aligned}
acq(sb) &\equiv \bigcup \{sb[k].A \mid k < |sb| \wedge (G(sb[k]) \vee vW(sb[k]))\} \\
rels(sb) &\equiv \bigcup \{sb[k].R \mid k < |sb| \wedge (G(sb[k]) \vee vW(sb[k]))\}.
\end{aligned}
$$

The set of all PT addresses acquired (resp. released) by all instructions in store buffer sb is defined as

$$
\begin{aligned}
acq_{pt}(sb) &\equiv \bigcup \{sb[k].A_{pt} \mid k < |sb| \wedge (G(sb[k]) \vee vW(sb[k]))\} \\
rels_{pt}(sb) &\equiv \bigcup \{sb[k].R_{pt} \mid k < |sb| \wedge (G(sb[k]) \vee vW(sb[k]))\}.
\end{aligned}
$$

The following predicate checks whether annotations of a given instruction I are safe with respect to a given state of the machine and a given thread ID i:

$$
\begin{aligned}
&safe\_annot(c_{sbh}, i, I) \equiv G(I) \vee vW(I) \ \rightarrow \\
&\quad I.A \subseteq shared \cup I.R_{pt} \cup \mathcal{O}_{[i]} \wedge I.L \subseteq I.A \wedge I.A \cap I.A_{pt} = \emptyset \wedge \\
&\quad I.A \cap I.R = \emptyset \wedge I.R \subseteq \mathcal{O}_{[i]} \wedge I.A_{pt} \subseteq shared \cup pt_{[i]} \cup I.R \wedge \\
&\quad I.A_{pt} \cap I.R_{pt} = \emptyset \wedge I.R_{pt} \subseteq pt_{[i]}.
\end{aligned}
$$

Another predicate collects some basic safety properties for the ownership transfer of instruction I, which are needed for reordering of this transfer after an SB step of other thread:

$$
\begin{aligned}
&safe\_otran(c_{sbh}, i, I) \equiv (vW(I) \vee RMW(I) \vee G(I)) \wedge \\
&\quad I.L \subseteq I.A \wedge I.R \subseteq \mathcal{O}_{[i]} \cup acq(sb_{[i]}) \wedge I.R_{pt} \subseteq pt_{[i]} \cup acq_{pt}(sb_{[i]}) \wedge \\
&\quad (\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset) \wedge \\
&\quad (\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset).
\end{aligned}
$$

Sets of temporaries used for reading by instructions in instruction list is , store buffer sb or ϑ are defined as

$$load_t(is) \equiv \bigcup\{is[k].t \mid k < |is| \wedge (R(is[k]) \vee RMW(is[k]))\}$$
$$load_t(sb) \equiv \bigcup\{sb[k].t \mid k < |sb| \wedge R(sb[k])\}$$

A store operation (D, f), where the function f maps temporaries to a value and D specifies the subset of temporaries, is valid iff f only depends on the temporaries specified by D:

$$valid\_sop((D, f)) \equiv \forall \vartheta.\ D \subseteq \mathrm{dom}(\vartheta) \rightarrow f(\vartheta) = f(\vartheta\ \restriction_D).$$

Ownership Invariants

1. oinv1: For every thread non-volatile writes in SB must refer to the owned memory. Reads in the suspended part of the SB have to be owned or refer to read-only memory. Note, that in the executed part of the SB reads do not always satisfy this property. Let $I = sb_{[i]}[k]$, then

$$oinv1(c_{sbh}) \equiv \forall i.\ \forall k < |sb_{[i]}|.\ (nvW(I) \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, i).\mathcal{O}_{[i]}) \wedge$$
$$(nvR(I) \wedge k \geq |exec(sb_{[i]})| \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, i).\mathcal{O}_{[i]} \cup \delta^k_{sb}(c_{sbh}, i).ro).$$

2. oinv2: Every outstanding volatile write is neither owned by any other thread and nor in other thread's PT set:

$$oinv2(c_{sbh}) \equiv \forall i.\ \forall I \in sb_{[i]}.\ vW(I) \rightarrow$$
$$I.pa \notin \bigcup_{j \neq i}(\mathcal{O}_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]})).$$

3. oinv3: In the suspended part of the store buffer outstanding accesses to read-only memory are not in the accumulated ownership sets of others. Note, that in the executed part of the SB reads do not always satisfy this property. Let $I = sb_{[i]}[k]$, then

$$oinv3(c_{sbh}) \equiv \forall i.\ \forall j \neq i.\ \forall k.\ k < |sb_{[i]}| \wedge k \geq |exec(sb_{[i]})| \wedge R(I) \wedge$$
$$I.pa \in \delta^k_{sb}(c_{sbh}, i).ro \rightarrow I.pa \notin (\mathcal{O}_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]})).$$

4. oinv4: The ownership sets of every two different threads are distinct:

$$oinv4(c_{sbh}) \equiv \forall i.\ \forall j \neq i.\ (\mathcal{O}_{[i]} \cup acq(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

Sharing Invariants

1. sinv1: All outstanding non-volatile writes are unshared. Let $I = sb_{[i]}[k]$, then

$$sinv1(c_{sbh}) \equiv \forall i.\ \forall k < |sb_{[i]}|.\ nvW(I) \rightarrow I.pa \notin \delta^k_{sb}(c_{sbh}, i).shared.$$

2. sinv2: All unshared addresses are owned or are in PT sets:

$$sinv2(c_{sbh}) \equiv \forall a \notin shared \rightarrow \exists i.\ a \in \mathcal{O}_{[i]} \cup pt_{[i]}.$$

3. sinv3: No thread owns read-only memory and read-only memory is shared:

$$sinv3(c_{sbh}) \equiv ro \subseteq shared \wedge \forall i.\ \mathcal{O}_{[i]} \cap ro = \emptyset.$$

4. sinv4: The ownership annotations of outstanding ghost and volatile write operations are consistent:

$$sinv4(c_{sbh}) = \forall i.\ \forall k < |sb_{[i]}|.\ safe\_annot(\delta^k_{sb}(c_{sbh}, i), i, sb_{[i]}[k]).$$

5. sinv5: There are no outstanding writes to read-only memory:

$$sinv5(c_{sbh}) \equiv \forall i.\ \forall k < |sb_{[i]}|.\ W(sb_{[i]}[k]) \rightarrow sb_{[i]}[k].pa \notin \delta^k_{sb}(c_{sbh}, i).ro.$$

Invariants on Temporaries

1. tinv1: The temporaries used for loads in the instruction list are distinct:

$$\begin{aligned} tinv1(c_{sbh}) \equiv \forall k < |is_{[i]}|.\ & load_t(is_{[i]}[0:k]) \cap \\ & load_t(is_{[i]}[k+1:|is_{[i]}|-1]) = \emptyset. \end{aligned}$$

2. tinv2: The temporaries used for loads in the store buffer are distinct:

$$\begin{aligned} tinv2(c_{sbh}) \equiv \forall k < |sb_{[i]}|.\ & load_t(sb_{[i]}[0:k]) \cap \\ & load_t(sb_{[i]}[k+1:|sb_{[i]}|-1]) = \emptyset. \end{aligned}$$

3. tinv3: The temporaries used for loads in an instruction list are fresh, i.e., are not in the domain of ϑ.

$$tinv3(c_{sbh}) \equiv \forall i.\ load_t(is_{[i]}) \cap \mathrm{dom}(\vartheta_{[i]}) = \emptyset.$$

Data Dependency Invariants

1. dinv1: Every store (D, f) in the instruction list or the store buffer is valid according to $valid\_sop$:

$$dinv1(c_{sbh}) \equiv \forall i.\ \forall I \in sb_{[i]} \circ is_{[i]}.\ (W(I) \vee RMW(I)) \rightarrow valid\_sop(I.(D, f))$$

2. dinv2: Domain D of a store instruction in the instruction list is a subset of previous read temporaries. Let $I = is_{[i]}[k]$, then

$$\begin{aligned} dinv2(c_{sbh}) \equiv \forall i.\ \forall k < |is_{[i]}|.\ (W(I) \vee RMW(I)) \rightarrow \\ I.D \subseteq dom(\vartheta_{[i]}) \cup load_t(is_{[i]}[0:k]) \end{aligned}$$

History Invariants

1. hinv1: In the suspended part of the SB the value stored for a non volatile read is the same as the last write to the same address in the SB or the value in memory, in case there is no hitting write in the buffer. Note, that in the executed part of the SB reads do not always satisfy this property. Let $I = sb_{[i]}[k]$, then

$$\begin{aligned} hinv1(c_{sbh}) \equiv \forall i.\ \forall k < |sb_{[i]}|.\ k \geq |exec(sb_{[i]})| \wedge nvR(I) \rightarrow \\ I.v = \delta^k_{sb}(c_{sbh}, i).m(I.pa) \end{aligned}$$

2. hinv2: There are no volatile reads in the suspended part of the store buffer:

$$hinv2(c_{sbh}) \equiv \forall i.\ \forall I \in susp(sb_{[i]}).\ \neg vR(I).$$

3. hinv3: For every read the recorded value coincides with the corresponding value in the temporaries:

$$hinv3(c_{sbh}) \equiv \forall i.\ \forall I \in (sb_{[i]}).\ R(I) \rightarrow I.v = \vartheta_{[i]}(I.t).$$

4. hinv4: For every write in a store buffer the recorded value v coincides with $f(\vartheta_{[i]})$ and domain D is a subset or previous read temporaries. Let $I = sb_{[i]}[k]$, then

$$\begin{aligned} hinv4(c_{sbh}) \equiv \forall i.\ \forall k < |sb_{[i]}|.\ W(I) \rightarrow I.f(\vartheta_{[i]}) = I.v \wedge \\ I.D \subseteq dom(\vartheta_{[i]}) \setminus load_t(sb[k+1:|sb_i|-1]). \end{aligned}$$

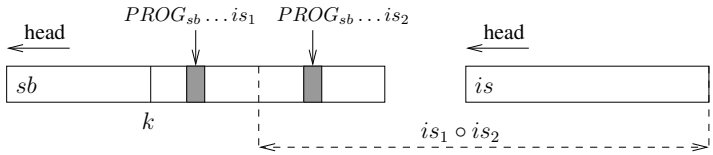

Fig. 12: Relating generated instructions with instructions in the store buffer and in the instruction list.

5. hinv5: History information for program steps in the store buffer is consistent. Let $I = sb_{[i]}[k], sb' = sb_{[i]}[k+1 : |sb_{[i]}| - 1]$, then

$$hinv5(c_{sbh}) \equiv \forall i.\ \forall k < |sb_{[i]}|.\ P(I) \rightarrow I.p2 = hd\text{-}p(p_{[i]}, sb') \wedge \\ \delta_p(I.p1, del\text{-}t(\vartheta_{[i]}, sb')) = (I.p2, I.is).$$

6. hinv6: Any suffix of the store buffer concatenated with the instruction list contains instructions generated by the program steps in this suffix (see Fig. 12). Let $sb' = sb[k : |sb_{[i]}| - 1]$, then

$$hinv6(c_{sbh}) \equiv \forall i.\ \forall k < |sb_{[i]}|.\ \exists is'.\ ins(sb') \circ is_{[i]} = is' \circ p\text{-}ins(sb').$$

MMU Invariant

1. minv1: In translated mode the physical address of an instruction in the store buffer is present in the current address translation set:

$$minv1(c_{sbh}) \equiv \forall i.\ \forall I \in sb_{[i]}.\ (R(I) \vee W(I)) \rightarrow \\ I.pa \in \text{atran}(mmu_{[i]}, I.va, mode_{[i]}, I.r).$$

Page Table Invariants

1. pinv1: Page table sets of different threads do not overlap:

$$pinv1(c_{sbh}) \equiv \forall i, j.\ i \neq j \rightarrow (pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset.$$

2. pinv2: Page table sets and ownership sets of different threads do not overlap:

$$pinv2(c_{sbh}) \equiv \forall i, j.\ i \neq j \rightarrow (pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

3. pinv3: Page table sets and the shared set do not overlap:

$$pinv3(c_{sbh}) \equiv \forall i.\ pt_{[i]} \cap shared = \emptyset.$$

4. pinv4: Page table set and ownership sets of one thread are disjoint:

$$pinv4(c_{sbh}) \equiv \forall i.\ pt_{[i]} \cap \mathcal{O}_{[i]} = \emptyset.$$

5.5 Assumptions on Program Steps

We introduce a number of assumptions on program steps, which guarantee that the read temporaries are always fresh for every new read instruction. Let $\delta_p(p_{[i]}, \vartheta_{[i]}) = (p', is')$, then

1. load temporaries in is' are distinct:

$$\forall k < |is'|.\ load_t(is'[0:k]) \cap load_t(is'[k+1:|is'|-1]) = \emptyset,$$

2. load temporaries in is' are distinct from load temporaries in $is_{[i]}$ and $\vartheta_{[i]}$

$$load_t(is') \cap (load_t(is_{[i]}) \cup dom(\vartheta_{[i]})) = \emptyset,$$

3. store instructions in is' are valid and their domains only depend on the previously generated load temporaries:

$$\forall k < |is'|.\ I = is'[k] \wedge (W(I) \vee RMW(I)) \rightarrow valid\_sop(I.(D, f)) \wedge I.D \subseteq load_t(is'[0:k]) \cup load_t(is_{[i]}) \cup \mathrm{dom}(\vartheta_{[i]}).$$

5.6 Proof Strategy

We split the proof of Theorem 1 into two parts. In the first part we assume safety of the delayed release and in step-by-step fashion show that the coupling invariant is maintained after every step of the SB machine.

Theorem 2 (Simulation).

$$c_{sbh} \Rightarrow c'_{sbh} \wedge c_{sbh} \sim c \wedge safe\text{-}reach_d(c) \wedge inv(c_{sbh}) \rightarrow inv(c'_{sbh}) \wedge (\exists c'.\ c \Rightarrow^* c' \wedge c'_{sbh} \sim c')$$

The proof of Theorem 2 is also split into two parts. In Sect. 6 we show that invariants are maintained after every step of the SB machine and in Sect. 7 we prove simulation itself. When proving that invariants and the coupling relation are maintained, we show only those properties, which can possibly get broken by the step. Those invariants and parts of the coupling relation which we don't consider explicitly are trivially maintained after the step. Note, that all invariants we defined talk about the content of the SB and trivially hold in the initial configuration, i.e. in the case when SBs of all threads are empty.

In the second part of the proof of Theorem 1 we show that safety of the delayed release can be derived from regular safety of the virtual machine.

Theorem 3 (safety).

$$initial(c) \wedge safe\text{-}reach(c) \rightarrow safe\text{-}reach_d(c)$$

This proof is given in Sect. 8.

6 Maintaining Invariants

In this section we show that the invariants are maintained after every step of the SB machine.

6.1 SB Steps

Lemma 1 (invariants maintained by δ_{sb}).

$$\forall i.\ inv(c_{sbh}) \rightarrow inv(\delta_{sb}(c_{sbh}, i))$$

Proof. We let $I = hd(sb_{[i]})$ and $c'_{sbh} = \delta_{sb}(c_{sbh}, i)$. From the semantics of the SB step we have

$$sb'_{[i]} = tl(sb_{[i]}) \quad \text{and} \quad is'_{[i]} = is_{[i]}.$$

We consider only the invariants which are affected by the SB step. For all invariants except $hinv6$ and $hinv1$ we only consider case $G(I) \vee vW(I)$. For $hinv1$ we only consider case $W(I)$.

- $oinv1$. Let $I = sb_{[j]}[k]$, then from $oinv1(c_{sbh})$ we have for all threads j

  $$\forall k < |sb_{[j]}|.\ (nvW(I) \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]}) \wedge (nvR(I) \wedge k \geq |exec(sb_{[j]})| \rightarrow I.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \cup \delta^k_{sb}(c_{sbh}, j).ro).$$

  For case $j = i$ the property is trivially maintained. For $j \neq i$ the first statement of the invariant also cannot be broken by a step of thread i. For the second statement we have to show for non-volatile reads $I^j = sb_{[j]}[k]$ that

  $$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro \rightarrow I^j.pa \in \delta^k_{sb}(c'_{sbh}, j).ro.$$

  From $oinv3(c_{sbh})$ that

  $$I^j.pa \notin (\mathcal{O}_{[i]} \cup acq(sb_{[i]}) \cup pt_{[i]} \cup acq_{pt}(sb_{[i]})).$$

  Hence,

  $$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro \rightarrow I^j.pa \notin I.A \cup I.A_{pt}$$

  $I^j.pa$ can not be acquired by thread i and remains in the read only set.
- $oinv2$. Invariant is obviously maintained because

  $$\mathcal{O}'_{[i]} \cup acq(sb'_{[i]}) \cup pt'_{[i]} \cup acq_{pt}(sb'_{[i]}) \subseteq \mathcal{O}_{[i]} \cup acq(sb_{[i]}) \cup pt_{[i]} \cup acq_{pt}(sb_{[i]}).$$

- $oinv3$. From $oinv3(c_{sbh})$ we have for all threads j:

  $$\forall j' \neq j.\ \forall k < |sb_{[j]}|.\ k \geq |exec(sb_{[j]})| \wedge R(I) \wedge I.pa \in \delta^k_{sb}(c_{sbh}, j).ro \rightarrow I.pa \notin (\mathcal{O}_{[j']} \cup acq(sb_{[j']}) \cup pt_{[j']} \cup acq_{pt}(sb_{[j']})).$$

For case $j = i$ the property is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and

$$R(I^j) \wedge I^j.pa \in \delta^k_{sb}(c'_{sbh}, j).ro.$$

From the semantics of SB steps we know that

$$\forall j'.\ \mathcal{O}'_{[j']} \cup acq(sb'_{[j']}) \cup pt'_{[j']} \cup acq_{pt}(sb'_{[j']}) \subseteq \mathcal{O}_{[j']} \cup acq(sb_{[j']}) \cup pt_{[j']} \cup acq_{pt}(sb_{[j']}).$$

Hence, all we have to show is

$$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro. \tag{1}$$

If $I^j.pa \notin c'_{sbh}.ro$, then it is released by thread j later and (1) trivially holds. If $I^j.pa \in c'_{sbh}.ro$ and $I^j.pa \notin c_{sbh}.ro$, then

$$I^j.pa \in I.R \subseteq \mathcal{O}_{[i]}.$$

From $oinv1(c_{sbh})$ we know that

$$I^j.pa \in \mathcal{O}_{[j]} \cup acq(sb_j) \cup \delta^k_{sb}(c_{sbh}, j).ro.$$

If $I^j.pa \in \mathcal{O}_{[j]} \cup acq(sb_j)$, we get a contradiction from $oinv4$. Hence, we can conclude

$$I^j.pa \in c_{sbh}.ro \vee I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro.$$

With $I^j.pa \in \delta^k_{sb}(c'_{sbh}, j).ro$, (1) obviously holds.

– $oinv4$. From $oinv4(c_{sbh})$ we have:

$$\forall j \neq i.\ (\mathcal{O}_{[i]} \cup acq(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset$$

From the definition of acq and semantics of the SB step, we have:

$$\begin{aligned} \mathcal{O}'_{[i]} &\subseteq \mathcal{O}_{[i]} \cup A \\ &\subseteq \mathcal{O}_{[i]} \cup acq(sb_{[i]}) \\ acq(sb'_{[i]}) &\subseteq acq(sb_{[i]}). \end{aligned}$$

Thus, we can conclude:

$$\forall j \neq i.\ (\mathcal{O}'_{[i]} \cup acq(sb'_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

Since the configuration of other threads is unchanged in c'_{sbh}, we get

$$oinv4(c'_{sbh}).$$

– $sinv1$. From $sinv1(c_{sbh})$ we have for all threads j

$$\forall k < |sb_{[j]}|.\ I = sb_{[j]}[k] \wedge nvW(I) \rightarrow I.pa \notin \delta^k_{sb}(c_{sbh}, j).shared.$$

For case $j = i$ the property is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $nvW(I^j)$. We have from $oinv1(c_{sbh})$

$$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \subseteq \mathcal{O}_{[j]} \cup acq(sb_{[j]}).$$

From $oinv4(c_{sbh})$ and $pinv2(c_{sbh})$ we can conclude:

$$(\mathcal{O}_{[i]} \cup acq(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset \wedge$$
$$(pt_{[i]} \cup acq(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset$$

From $sinv4(c_{sbh})$ and the semantics of SB steps we get

$$\begin{aligned}\delta^k_{sb}(c'_{sbh}, j).shared &\subseteq \delta^k_{sb}(c_{sbh}, j).shared \cup I.R \cup I.R_{pt} \\ &\subseteq \delta^k_{sb}(c_{sbh}, j).shared \cup \mathcal{O}_{[i]} \cup pt_{[i]}.\end{aligned}$$

Hence, the step of thread i can not make address $I^j.pa$ shared and the invariant is maintained.

– $sinv2$. From $sinv2(c_{sbh})$ we have

$$\forall a \notin shared \rightarrow \exists j.\ a \in \mathcal{O}_{[j]} \cup pt_{[j]}.$$

We do a case split:

- $a \notin shared \wedge a \notin shared'$. Hence,

$$\exists j.\ a \in \mathcal{O}_{[j]} \cup pt_{[j]}.$$

If $j \neq i$, then the statement is trivially maintained. If $j = i$, we assume

$$a \notin \mathcal{O}'_{[j]} \cup pt'_{[j]}$$

and prove by contradiction.

  * if $a \in \mathcal{O}_{[i]}$, then we have from the semantics and from $sinv4(c_{sbh})$

$$a \in I.R \wedge a \notin I.A \wedge a \notin I.L \wedge a \notin I.A_{pt},$$

which implies $a \in shared'$ and gives a contradiction.

  * if $a \in pt_{[i]}$, then

$$a \in I.R_{pt} \wedge a \notin I.A_{pt} \wedge a \notin I.A \wedge a \notin I.L,$$

which again implies $a \in shared'$ and gives a contradiction.

- $a \in shared \wedge a \notin shared'$. Using $sinv4(c_{sbh})$ we get

$$a \in I.L \cup I.A_{pt} \subseteq I.A \cup I.A_{pt} \subseteq \mathcal{O}'_{[j]} \cup pt'_{[j]}.$$

– $sinv3$. From $sinv3(c_{sbh})$, we have

$$\forall j.\ \mathcal{O}_{[j]} \cap ro = \emptyset \wedge ro \subseteq shared.$$

From the semantics, we have:

$$ro' = ro \cup (I.R \setminus I.W) \setminus (I.A \cup I.A_{pt}),$$
$$\mathcal{O}'_{[i]} = \mathcal{O}_{[i]} \cup I.A \setminus I.R.$$

Thus, we can conclude:

$$ro' \cap \mathcal{O}'_{[i]} = \emptyset.$$

From $sinv4(c_{sbh})$ and $oinv4(c_{sbh})$ we get

$$\forall j \neq i.\ I.R \cap \mathcal{O}_{[j]} = \emptyset,$$

which implies

$$\forall j.\ \mathcal{O}'_{[j]} \cap ro' = \emptyset.$$

For the shared set we have from the semantics

$$shared' = shared \cup I.R \cup I.R_{pt} \setminus (I.L \cup I.A_{pt})$$

From $sinv4(c_{sbh})$, we can conclude:

$$I.L \subseteq I.A,$$

which implies

$$ro' \subseteq shared'.$$

– $sinv4$. From $sinv4(c_{sbh})$ we have

$$\forall j.\ \forall k < |sb_{[j]}|.\ safe\_annot(\delta^k_{sb}(c_{sbh}, j), j, sb_{[j]}[k]).$$

For case $j = i$ the invariant is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $vW(I^j) \vee G(I^j)$. The local ownership sets of thread j remain unchanged. Hence, all we have to show is

$$I^j.A \cap \delta^k_{sb}(c_{sbh}, j).shared = I^j.A \cap \delta^k_{sb}(c'_{sbh}, j).shared, \tag{2}$$
$$I^j.A_{pt} \cap \delta^k_{sb}(c_{sbh}, j).shared = I^j.A_{pt} \cap \delta^k_{sb}(c'_{sbh}, j).shared. \tag{3}$$

We first show (2) Let

$$a \in I^j.A \cap \delta^k_{sb}(c_{sbh}, j).shared.$$

From the semantics of SB steps we have

$$\delta^k_{sb}(c'_{sbh}, j).shared = \delta^k_{sb}(c_{sbh}, j).shared \cup I.R \cup I.R_{pt} \setminus (I.L \cup I.A_{pt}).$$

From $sinv4(c_{sbh})$, $oinv4(c_{sbh})$ and $pinv2(c_{sbh})$ we get

$$I^j.A \cap (I.L \cup I.A_{pt}) = \emptyset$$
$$I^j.A \cap (I.R \cup I.R_{pt}) = \emptyset.$$

And we conclude (2). The proof of (3) is completely analogous if one takes $pinv1(c_{sbh})$ instead of $oinv4(c_{sbh})$.

– $sinv5(c_{sbh})$. From $sinv5$ we have

$$\forall j.\ \forall k < |sb_{[j]}|.\ W(sb_{[j]}[k]) \rightarrow sb_{[j]}[k].pa \notin \delta^k_{sb}(c_{sbh}, j).ro.$$

For case $j = i$ the invariant is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $W(I^j)$. We have from the semantics of SB steps and $sinv4$

$$\begin{aligned}\delta^k_{sb}(c'_{sbh}, j).ro &\subseteq \delta^k_{sb}(c_{sbh}, j).ro \cup I.R \\ &\subseteq \delta^k_{sb}(c_{sbh}, j).ro \cup \mathcal{O}_{[i]}.\end{aligned}$$

We consider cases:

- $nvW(I^j)$. With $oinv1(c_{sbh})$ we have

$$I^j.pa \in \mathcal{O}_{[j]} \cup acq(sb_{[j]}).$$

From $oinv4(c_{sbh})$ we conclude

$$\mathcal{O}_{[i]} \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

Hence,

$$I^j.pa \notin \delta^k_{sb}(c'_{sbh}, j).ro.$$

- $vW(I^j)$. The proof immediately follows from $oinv2$.

– $hinv1(c_{sbh})$. From $hinv1(c_{sbh})$ we have for all j and for all $k < |sb_{[j]}|$:

$$k \geq |exec(sb_{[j]})| \wedge I = sb_{[j]}[k] \wedge nvR(I) \rightarrow I.v = \delta^k_{sb}(c_{sbh}, j).m(I.pa).$$

For case $j = i$ the invariant is trivially maintained. For $j \neq i$ let $I^j = sb_{[j]}[k]$ and $nvR(I^j)$. If $I^j.v \neq c_{sbh}.m(I^j.pa)$, then there is a write to $I^j.pa$ in $sb_{[k]}[0 : k-1]$, which is executed before the read. Hence, the step of thread i cannot possibly break the invariant. If $I^j.v = c_{sbh}.m(I^j.pa)$, then from $oinv1(c_{sbh})$ we have

$$I^j.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \cup \delta^k_{sb}(c_{sbh}, j).ro.$$

The only step of thread i which can change the memory content is $W(I)$. We now do a case split on $I^j.pa$ and show that $I.pa \neq I^j.pa$.

- $I^j.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]}$. Hence,

$$I^j.pa \in \mathcal{O}_{[j]} \cup acq(sb_{[j]}).$$

From $oinv1(c_{sbh})$, $oinv2(c_{sbh})$ and $oinv4(c_{sbh})$ we conclude:

$$W(I) \rightarrow I.pa \notin \bigcup_{\forall j \neq i} (\mathcal{O}_{[j]} \cup acq(sb_{[j]})),$$

which implies $I.pa \neq I^j.pa$

- $I^j.pa \in \delta^k_{sb}(c_{sbh}, j).ro$. In this case we do a further case split on I.

∗ $nvW(I)$. From $oinv3(c_{sbh})$ we get

$$I^j.pa \notin (\mathcal{O}_{[i]} \cup acq(sb_{[i]}) \cup pt_{[i]} \cup acq_{pt}(sb_{[i]})).$$

From $oinv1(c_{sbh})$ we have

$$I.pa \in \mathcal{O}_{[i]} \cup acq(sb_{[i]}).$$

Hence, $I.pa \neq I^j.pa$

∗ $vW(I)$. If $I^j.pa \in c_{sbh}.ro$, from $sinv5(c_{sbh})$ we can get

$$I.pa \notin c_{sbh}.ro.$$

If $I^j \notin c_{sbh}.ro$, we can conclude

$$I^j.pa \in \mathcal{O}_{[j]} \cup acq(sb_{[j]}).$$

From $oinv2(c_{sbh})$, we get $I.pa \neq I^j.pa$.

Finally, we can get

$$\delta^k_{sb}(c_{sbh}, j).m(I^j.pa) = \delta^k_{sb}(c'_{sbh}, j).m(I^j.pa)$$

and concludes the proof.

– $hinv6$. From $hinv6(c_{sbh})$ we have for all $k < |sb_{[i]}|$:

$$\exists is.\ ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} = is \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]).$$

For $k' < |sb'_{[i]}|$ we get for all prefixes is:

$$ins(sb'_{[i]}[k' : |sb'_{[i]}| - 1]) \circ is'_{[i]} = ins(sb_{[i]}[k' + 1 : |sb_{[i]}| - 1]) \circ is_{[i]}$$
$$is \circ p\text{-}ins(sb'_{[i]}[k' : |sb'_{[i]}| - 1]) = is \circ p\text{-}ins(sb_{[i]}[k' + 1 : |sb_{[i]}| - 1]).$$

Hence, we instantiate $hinv6(c_{sbh})$ with $k'+1$ and get the proof for $hinv6(c'_{sbh})$.

– $pinv1$. From $pinv1(c_{sbh})$, we have:

$$\forall j \neq i.\ (pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset.$$

From the definition of acq_{pt} and the semantics of the SB step, we have:

$$pt'_{[i]} \cup acq_{pt}(sb'_{[i]}) \subseteq pt_{[i]} \cup acq_{pt}(sb_{[i]}).$$

Since the configuration of other threads is unchanged in c'_{sbh}, we get

$$pinv1(c'_{sbh})$$

– $pinv2$. From $pinv2(c_{sbh})$, we have:

$$\forall i. \forall j \neq i. (pt_{[i]} \cup acq_{pt}(sb_{[i]})) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

We conclude

$$pt'_{[i]} \cup acq_{pt}(sb'_{[i]}) \subseteq pt_{[i]} \cup acq_{pt}(sb_{[i]})$$
$$\mathcal{O}'_{[i]} \cup acq(sb'_{[i]}) \subseteq \mathcal{O}_{[i]} \cup acq(sb_{[i]}),$$

which implies $pinv2(c'_{sbh})$.

– $pinv3$. From $pinv3(c_{sbh})$, we have:

$$\forall j.\ pt_{[j]} \cap shared = \emptyset.$$

From the semantics we get

$$pt'_{[i]} = pt_{[i]} \cup I.A_{pt} \setminus I.R_{pt}$$
$$shared' \subseteq shared \cup I.R \cup I.R_{pt} \setminus I.A_{pt}.$$

From $sinv4(c_{sbh})$ and $pinv4(c_{sbh})$ we know that

$$pt_{[i]} \cap I.R = \emptyset.$$

Hence, all new addresses which are added to the shared set are not present in $pt'_{[i]}$. Addresses $I.A_{pt}$ which are added to the pt set, are excluded from the shared set. Therefore, we have

$$pt'_{[i]} \cap shared' = \emptyset.$$

For $j \neq i$ we have from $sinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$

$$pt_{[j]} \cap I.R = pt_{[j]} \cap I.R_{pt} = \emptyset.$$

Thus,

$$pt'_{[j]} \cap shared' = \emptyset.$$

– $pinv4$. From $pinv4(c_{sbh})$, we have:

$$pt_{[i]} \cap \mathcal{O}_{[i]} = \emptyset.$$

For $G(I) \vee vW(I)$ we get

$$pt'_{[i]} \subseteq pt_{[i]} \cup I.A_{pt}$$
$$\mathcal{O}'_{[i]} \subseteq \mathcal{O}_{[i]} \cup I.A.$$

From $sinv4(c_{sbh})$ and $pinv2(c_{sbh})$ we have

$$pt_{[i]} \cap I.A = \mathcal{O}_{[i]} \cap A_{pt} = I.A_{pt} \cap I.A = \emptyset,$$

and conclude the proof.

□

6.2 Commutativity of SB Steps

The following function applies the ownership transfer of instruction I in thread i to the provided configuration of the SB machine:

$$otran\text{-}sbh(c_{sbh}, i, I) = c_{sbh}[ghst[i] := otran(c_{sbh}.ghst[i], i, I)].$$

Lemma 2 (ownership transfer commute).

$$inv(c_{sbh}) \wedge safe\_otran(c_{sbh}, i, I) \wedge i \neq j \rightarrow$$
$$\delta_{sb}(otran\text{-}sbh(c_{sbh}, i, I), j) = otran\text{-}sbh(\delta_{sb}(c_{sbh}, j), i, I)$$

Proof. The case $|sb_{[j]}| = 0$ is trivial. Otherwise, let I^j denote the first instruction in $sb_{[j]}$:

$$I^j = sb_{[j]}[0].$$

For $I^j.A, I^j.R, \ldots$ we abbreviate $A^j, R^j, \ldots$ and for $I.A, I.R, \ldots$ we write $A, R, \ldots$. We set

$$c'_{sbh} = \delta_{sb}(otran\text{-}sbh(c_{sbh}, i, I), j) \quad \text{and} \quad c''_{sbh} = otran\text{-}sbh(\delta_{sb}(c_{sbh}, j), i, I).$$

– For components $c_{sbh}.X$, where $X \in \{shared, ro\}$ we only have to consider cases when $G(I^j) \vee vW(I^j)$. From definitions of δ_{sb} and the ownership transfer we have:

$$c'_{sbh}.shared = shared \cup R_{pt} \cup R \setminus (L \cup A_{pt}) \cup R^j_{pt} \cup R^j \setminus (L^j \cup A^j_{pt})$$
$$c'_{sbh}.ro = ro \cup (R \setminus W) \setminus (A \cup A_{pt}) \cup (R^j \setminus W^j) \setminus (A^j \cup A^j_{pt}).$$

From $sinv4(c_{sbh})$ and definitions of acq and acq_{pt} we can conclude

$$L^j \subseteq A^j \wedge (A^j \cup A^j_{pt}) \subseteq \mathcal{O}_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]}) \wedge \quad (4)$$
$$(R^j \cup R^j_{pt}) \subseteq \mathcal{O}_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]}).$$

From $oinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we know that sets $\mathcal{O}_{[j]} \cup acq(sb_{[j]})$ and $pt_{[j]} \cup acq_{pt}(sb_{[j]})$ don't overlap with the accumulated ownership sets of thread i. Predicate $safe\_otran(c_{sbh}, i, I)$ guarantees that release sets of instruction I don't overlap with the accumulated ownership sets of thread j and acquire sets of instruction I are subsets of the accumulated ownership sets of thread i. Hence we can conclude that acquire and release sets of instructions I and I^j don't overlap:

$$(L \cup A \cup A_{pt}) \cap (R^j \cup R^j_{pt}) = \emptyset \quad (5)$$
$$(L^j \cup A^j \cup A^j_{pt}) \cap (R \cup R_{pt}) = \emptyset.$$

Hence,

$$\begin{aligned} c'_{sbh}.shared &= shared \cup R_{pt} \cup R \cup R^j_{pt} \cup R^j \setminus (L \cup A_{pt} \cup L^j \cup A^j_{pt}) \\ &= shared \cup R^j_{pt} \cup R^j \cup R_{pt} \cup R \setminus (L^j \cup A^j_{pt} \cup L \cup A_{pt}) \\ &= c''_{sbh}.shared \\ c'_{sbh}.ro &= ro \cup (R \setminus W) \cup (R^j \setminus W^j) \setminus (A \cup A_{pt}) \setminus (A^j \cup A^j_{pt}) \\ &= ro \cup (R^j \setminus W^j) \cup (R \setminus W) \setminus (A^j \cup A^j_{pt}) \setminus (A \cup A_{pt}) \\ &= c''_{sbh}.ro. \end{aligned}$$

– For thread local configurations $c_{sbh}.ts$ only the release sets might get affected by the reordering in case $G(I^j) \vee vW(I^j)$. We first consider case $G(I) \wedge G(I^j)$. For the shared release set we have

$$\begin{aligned}
c'_{sbh}.rls_{s[i]} &= rls_{s[i]} \cup (R \cap shared) \\
c''_{sbh}.rls_{s[i]} &= rls_{s[j]} \cup (R \cap (shared \cup R^j \cup R^j_{pt} \setminus (L^j \cup A^j_{pt}))) \\
c''_{sbh}.rls_{s[j]} &= rls_{s[j]} \cup (R^j \cap shared) \\
c'_{sbh}.rls_{s[j]} &= rls_{s[i]} \cup (R^j \cap (shared \cup R \cup R_{pt} \setminus (L \cup A_{pt}))).
\end{aligned}$$

From (4), $safe\_otran(c_{sbh}, i, I)$, $oinv4(c_{sbh})$ and $pinv2(c_{sbh})$ we know that release sets of different threads are disjoint:

$$(R \cup R_{pt}) \cap (R^j \cup R^j_{pt}) = \emptyset.$$

Hence, with (5) we have

$$\begin{aligned}
c'_{sbh}.rls_{s[j]} &= rls_{s[j]} \cup (R^j \cap shared) \\
&= c''_{sbh}.rls_{s[j]} \\
c''_{sbh}.rls_{s[i]} &= rls_{s[i]} \cup (R \cap shared) \\
&= c'_{sbh}.rls_{s[i]}.
\end{aligned}$$

For case $(vW(I) \vee RMW(I)) \wedge G(I^j)$ we conclude

$$\begin{aligned}
c'_{sbh}.rls_{s[j]} &= rls_{s[j]} \cup (R^j \cap (shared \cup R \cup R_{pt} \setminus (L \cup A_{pt}))) \\
&= rls_{s[j]} \cup (R^j \cap shared) \\
&= c''_{sbh}.rls_{s[j]} \\
c''_{sbh}.rls_{s[i]} &= c'_{sbh}.rls_{s[i]} = \emptyset.
\end{aligned}$$

For case $(vW(I) \vee RMW(I)) \wedge vW(I^j)$ we obviously get

$$\begin{aligned}
c'_{sbh}.rls_{s[i]} &= c''_{sbh}.rls_{s[i]} = \emptyset \\
c'_{sbh}.rls_{s[j]} &= c''_{sbh}.rls_{s[j]} = \emptyset.
\end{aligned}$$

For case $G(I) \wedge vW(I^j)$ we conclude

$$\begin{aligned}
c'_{sbh}.rls_{s[i]} &= rls_{s[i]} \cup (R \cap shared) \\
&= rls_{s[i]} \cup (R \cap (shared \cup R^j \cup R^j_{pt} \setminus (L^j \cup A^j_{pt}))) \\
&= c''_{sbh}.rls_{s[i]} \\
c''_{sbh}.rls_{s[j]} &= c'_{sbh}.rls_{s[j]} = \emptyset.
\end{aligned}$$

The proof for the equality of the local and page table release sets is completely analogous.

□

Lemma 3 (ownership transfer safe for SB instruction).

$$inv(c_{sbh}) \rightarrow safe\_otran(c_{sbh}, i, hd(sb_{[i]}))$$

Proof. The proof immediately follows from $sinv4(c_{sbh})$, $oinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ as well as definition of acq and acq_{pt}. □

Lemma 4 (δ_{sb} commute).

$$inv(c_{sbh}) \wedge \neg vW(hd(sb_{[i]})) \rightarrow \delta_{sb}(\delta_{sb}(c_{sbh}, i), j) = \delta_{sb}(\delta_{sb}(c_{sbh}, j), i)$$

Proof. The case when one of the SBs is empty or $i = j$ is trivial. Otherwise, for $k \in \{i, j\}$ let I^k denote the first instruction in $sb_{[k]}$:

$$I^k = sb_{[k]}[0].$$

We set

$$c'_{sbh} = \delta_{sb}(\delta_{sb}(c_{sbh}, i), j) \quad \text{and} \quad c''_{sbh} = \delta_{sb}(\delta_{sb}(c_{sbh}, j), i).$$

With Lemma 3 we conclude

$$safe\_otran(c_{sbh}, i, I^i).$$

Applying Lemma 2 we get the equality of all ownership and release sets in configurations c'_{sbh} and c''_{sbh}. Hence, the only part which is left to show is the equality of the memory component. The only interesting case here is $nvW(I^i) \wedge W(I^j)$. We show that $I^i.pa \neq I^j.pa$. From $oinv1(c_{sbh})$ we can conclude:

$$I^i.pa \in \mathcal{O}_{[i]}.$$

We now do a case distinctions on I^j. In case $nvW(I^j)$ we conclude from $oinv1(c_{sbh})$ and $oinv4(c_{sbh})$

$$I^j.pa \in \mathcal{O}_{[j]} \quad \text{and} \quad I^j.pa \notin \mathcal{O}_{[i]}.$$

In case $vW(I^j)$ we use $oinv2(c_{sbh})$ to directly conclude:

$$I^j.pa \notin \mathcal{O}_{[i]}.$$

□

Lemma 5 ($\delta^k_{sb}, \Delta^{exec}_{sb}$ commute).

$$\forall k \leq |sb_{[i]}|.\ inv(c_{sbh}) \rightarrow \delta^k_{sb}(\Delta^{exec}_{sb}(c_{sbh}, j), i) = \Delta^{exec}_{sb}(\delta^k_{sb}(c_{sbh}, i), j)$$

Proof. For case $|exec(sb_{[j]})| = 0$ or $k = 0$ the proof is trivial. Otherwise, let

$$c'_{sbh} = \delta_{sb}(\Delta^{exec}_{sb}(c_{sbh}, j), i).$$

Lemma 1 guarantees that invariants are maintained by any number of SB steps. Applying Lemma 4 we can reorder the last step of thread j after the first step of thread i:

$$c'_{sbh} = \delta_{sb}(\delta_{sb}(\Delta_{sb}^{|exec(sb_{[j]})-1|}(c_{sbh}, j), i), j).$$

Performing the same action $|exec(sb_{[j]})|$ times we move the step of thread i before all steps of thread j:

$$c'_{sbh} = \Delta_{sb}^{exec}(\delta_{sb}(c_{sbh}, i), j).$$

To reorder all k steps of thread i we have to repeat this procedure k times, resulting in $k \times |exec(sb_{[j]})|$ applications of lemma 4. □

Lemma 6 (Δ_{sb}^{exec} commute).

$$\begin{aligned}\forall i.\ inv(c_{sbh}) \rightarrow\ & \Delta_{sb}^{exec}(c_{sbh}) = \Delta_{sb}^{exec}(\Delta_{sb}^{exec}(c_{sbh}, i)) \wedge \\ & \Delta_{sb}^{exec}(c_{sbh}) = \Delta_{sb}^{exec}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i) \wedge \\ & \Delta_{sb}^{exec}(c_{sbh}) = \Delta_{sb[\neq i]}^{exec}(\Delta_{sb}^{exec}(c_{sbh}, i)) \wedge \\ & \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i) = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)) \wedge \\ & \forall k \leq |sb_{[i]}|.\ \delta_{sb}^{k}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i) = \Delta_{sb[\neq i]}^{exec}(\delta_{sb}^{k}(c_{sbh}, i))\end{aligned}$$

Proof. The proof follows directly from lemmas 5 and 1 and we omit the easy, but boring bookkeeping here □

6.3 Program Step

Lemma 7 (invariants maintained by program step).

$$inv(c_{sbh}) \wedge c_{sbh} \xRightarrow{p}_i c'_{sbh} \rightarrow inv(c'_{sbh})$$

Proof. From the semantics, we have $sb'_{[i]} = sb_{[i]} \circ PROG\ p_{[i]}\ p'_{[i]}\ is'$.

- Invariants dealing with temporaries (i.e., $tinv1$, $tinv3$, $dinv1$, $dinv2$) are easily maintained using the assumptions on the program state.
- $hinv5$. From $hinv5(c_{sbh})$ we have

  $$\begin{aligned}\forall k < |sb_{[i]}|.\ P(I) \rightarrow I.p2 = hd\text{-}p(p_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \wedge \\ \delta_p(I.p1, del\text{-}t(\vartheta_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}| - 1])) = (I.p2, I.is).\end{aligned}$$

  where $I = sb_{[i]}[k]$. For the newly recorded program step we have

  $$p'_{[i]} = hd\text{-}p(p'_{[i]}, [])\ \wedge \delta_p(p_{[i]}, \vartheta) = (p'_{[i]}, is'),$$

  and the required property holds. Since no new read instruction are added to the store buffer, we have $\vartheta = \vartheta'$ and

  $$\forall k < |sb_{[i]}|.\ load_t(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) = load_t(sb'_{[i]}[k+1 : |sb'_{[i]}| - 1]).$$

  Hence, the second statement of the invariant is maintained for all program steps, that were in the store buffer before the step. We now consider cases:

- case I is the last program instruction in $sb_{[i]}$. Then we have

$$\begin{aligned} I.p2 &= \mathit{hd\text{-}p}(p_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}|-1]) \\ &= p_{[i]} \\ &= \mathit{hd\text{-}p}(p'_{[i]}, sb'_{[i]}[k+1 : |sb'_{[i]}|-1]). \end{aligned}$$

- case I is not the last program instruction in $sb_{[i]}$. Then we have

$$\begin{aligned} I.p2 &= \mathit{hd\text{-}p}(p_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}|-1]) \\ &= \mathit{hd\text{-}p}(p'_{[i]}, sb_{[i]}[k+1 : |sb_{[i]}|-1]) \\ &= \mathit{hd\text{-}p}(p'_{[i]}, sb'_{[i]}[k+1 : |sb'_{[i]}|-1]). \end{aligned}$$

This concludes the proof for $\mathit{hinv}5$.

– $\mathit{hinv}6$. From $\mathit{hinv}6(c_{sbh})$ we have for all $k < |sb_{[i]}|$:

$$\exists is.\ \mathit{ins}(sb_{[i]}[k : |sb_{[i]}|-1]) \circ is_{[i]} = is \circ \mathit{p\text{-}ins}(sb_{[i]}[k : |sb_{[i]}|-1]).$$

After adding a program step to the store buffer we have for all $k < |sb_{[i]}|$:

$$\begin{aligned} \mathit{ins}(sb'_{[i]}[k : |sb'_{[i]}|-1]) &= \mathit{ins}(sb_{[i]}[k : |sb_{[i]}|-1]) \\ is'_{[i]} &= is_{[i]} \circ is' \\ \mathit{p\text{-}ins}(sb'_{[i]}[k : |sb'_{[i]}|-1]) &= \mathit{p\text{-}ins}(sb_{[i]}[k : |sb_{[i]}|-1]) \circ is', \end{aligned}$$

and the invariant holds if we choose the same prefix is, as we had before the step. For $k = |sb_{[i]}|$ we have

$$\begin{aligned} \mathit{ins}(sb'_{[i]}[k : |sb'_{[i]}|-1]) &= [] \\ is'_{[i]} &= is_{[i]} \circ is' \\ \mathit{p\text{-}ins}(sb'_{[i]}[k : |sb'_{[i]}|-1]) &= is', \end{aligned}$$

and the invariant holds if we set $is = is_{[i]}$. This concludes the proof for $\mathit{hinv}6$.

□

6.4 Memory Steps

In case of FENCE, INVLPG, mode switch and write to PTO memory steps the store buffer of thread i is empty. Hence, invariant $\mathit{minv}1$ is trivially maintained. Other invariants can not possibly be broken by the step.

RMW

Lemma 8 (safe execution maintains disjoint sets).

$$\mathit{disjoint\text{-}osets}(c) \wedge c \Rightarrow^* c' \wedge \mathit{safe\text{-}reach}(c) \rightarrow \mathit{disjoint\text{-}osets}(c')$$

Proof. We prove this lemma by induction on the length of the computation. If $c = c'$ there is nothing to prove. Otherwise, we assume

$$\mathit{disjoint\text{-}osets}(c) \wedge \mathit{safe\text{-}reach}(c)$$

as the induction hypothesis and have to prove

$$\forall c', i.\ c \Rightarrow_i c' \rightarrow \mathit{disjoint\text{-}osets}(c').$$

If we do not perform the ownership transfer, it is trivially true. Otherwise, we assume that $I = hd(c.is_{[i]})$ performs the ownership transfer. From $\mathit{safe\text{-}reach}(c)$ we conclude $\mathit{safe\text{-}state}(c)$ which infers:

$$I.L \subseteq I.A.$$

With the definition of the ownership transfer we have:

$$\begin{aligned} c'.ro &= c.ro \cup (I.R \setminus I.W) \setminus (I.A \cup I.A_{pt}) \\ c'.shared &= c.shared \cup I.R \cup I.R_{pt} \setminus (I.L \cup I.A_{pt}). \end{aligned}$$

With the induction hypothesis we can conclude:

$$c'.ro \subseteq c'.shared.$$

From $\mathit{disjoint\text{-}osets}(c)$ we trivially get for all $j \neq i$ and $k \neq i$

$$\begin{aligned} j \neq k \rightarrow c'.\mathcal{O}_{[k]} \cap c'.\mathcal{O}_{[j]} = \emptyset \ \wedge\ c'.\mathcal{O}_{[k]} \cap c'.pt_{[j]} = \emptyset \\ c'.pt_{[k]} \cap c'.pt_{[j]} = \emptyset \ \wedge\ c'.\mathcal{O}_{[k]} \cap c'.pt_{[k]} = \emptyset. \end{aligned}$$

From $\mathit{safe\text{-}state}(c)$ we have:

$$I.R \subseteq c.\mathcal{O}_{[i]} \ \wedge\ I.R_{pt} \subseteq c.pt_{[i]}$$

which gives us

$$\forall k \neq i.\ c'.\mathcal{O}_{[k]} \cap c'.ro = \emptyset \ \wedge\ c'.pt_{[k]} \cap c'.shared = \emptyset.$$

From the definition of the ownership transfer we also have:

$$\begin{aligned} c'.\mathcal{O}_{[i]} &= c.\mathcal{O}_{[i]} \cup I.A \setminus I.R \\ c'.pt_{[i]} &= c.pt_{[i]} \cup I.A_{pt} \setminus I.R_{pt}. \end{aligned}$$

From $\mathit{safe\text{-}state}(c)$, we have:

$$\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (c.\mathcal{O}_{[j]} \cup c.pt_{[j]}) = \emptyset.$$

With the induction hypothesis we can conclude:

$$\begin{aligned} \forall j \neq i.\ c'.\mathcal{O}_{[i]} \cap c'.\mathcal{O}_{[j]} = \emptyset \ \wedge\ c'.\mathcal{O}_{[i]} \cap c'.pt_{[j]} = \emptyset \ \wedge \\ c'.pt_{[i]} \cap c'.pt_{[j]} = \emptyset. \end{aligned}$$

From $safe\text{-}state(c)$ we also have:

$$I.A_{pt} \cap I.A = \emptyset.$$

Thus, with the induction hypothesis we can conclude:

$$c'.\mathcal{O}_{[i]} \cap c'.pt_{[i]} = \emptyset.$$

With the definition of ownership transfer we can also conclude:

$$c'.\mathcal{O}_{[i]} \cap c'.ro = \emptyset \ \wedge \ c'.pt_{[i]} \cap c'.shared = \emptyset.$$

□

Lemma 9 (coupling implies disjoint sets).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \rightarrow disjoint\text{-}osets(c)$$

Proof. Lemma 1 implies

$$inv(\Delta_{sb}^{exec}(c_{sbh})).$$

The statement of the lemma follows immediately from the coupling relations and from invariants $oinv4$, $pinv1$, $pinv2$, $pinv3$, $pinv4$, and $sinv3$. □

Lemma 10 (invariants maintained by RMW).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \stackrel{m}{\Longrightarrow}_i c'_{sbh} \ \wedge$$
$$hd(is_{[i]}) = \textbf{RMW} \text{ va t } (\mathrm{D}, \mathrm{f}) \text{ r cond } (\mathrm{A}, \mathrm{L}, \mathrm{R}, \mathrm{W}, \mathrm{A_{pt}}, \mathrm{R_{pt}}) \rightarrow inv(c'_{sbh})$$

Proof. Since the store buffer of thread i is empty when we perform the step, invariant $hinv6$ is trivially maintained. If RMW test fails, then all other invariants are trivially maintained. Invariants which might get broken by the RMW step if the test succeeds are considered below.

– $oinv1$. We proceed the same way as in the proof of $oinv1$ in Lemma 1. We have to show for $j \neq i$ for all non-volatile reads $I = sb_{[j]}[k]$ in the suspended part of the SB, that

$$I.pa \in \delta_{sb}^{k}(c_{sbh}, j).ro \rightarrow I.pa \notin A \cup A_{pt}. \tag{6}$$

Let c' be configuration of the abstract machine after we execute the RMW step of thread i:

$$c \stackrel{\mathrm{m}}{\Longrightarrow}_i c'.$$

The reasons why this step can be executed and why the result of the RMW test is the same as in the SB machine are given in the proof of lemma 33. After the step we obviously have

$$A \subseteq c'.\mathcal{O}_{[i]} \quad \text{and} \quad A_{pt} \subseteq c'.pt_{[i]}.$$

Let n be the number of instructions up to instruction k in the suspended part of store buffer j:

$$n = k - |exec(sb_{[j]})|.$$

We execute n steps of thread j in the abstract machine starting from configuration c'. The choice of the steps to be executed (program or memory) depends on the type of the store buffer instructions under consideration. We refer to the resulting configuration as c'':

$$c' \overset{\text{p,m}}{\Longrightarrow}{}^{n}_{j} c''.$$

All the instructions in the sequence can be executed in the abstract machine, because from $minv1$ and the coupling invariant we have for all $I \in sb_{[j]}$:

$$(R(I) \vee W(I)) \rightarrow I.pa \in \text{atran}(c.mmu_{[j]}, I.va, c.mode_{[j]}, I.r).$$

and there are no instructions recorded in SBs, which affect the MMU state or the translation mode. Since the steps of thread j do not affect the ownership sets of thread i we still have

$$A \subseteq c''.\mathcal{O}_{[i]} \quad \text{and} \quad A_{pt} \subseteq c''.pt_{[i]}.$$

Since c'' is a configuration reachable from c it follows

$$safe\text{-}reach_d(c'').$$

Hence, instruction I in thread j still has to be safe, which implies

$$I.pa \in c''.\mathcal{O}_{[j]} \cup c''.ro.$$

With lemmas 9 and 8 we conclude (6).

– $oinv2$. Let $I = sb_{[j]}[k]$ be a volatile write in store buffer j. To maintain the invariant after the step of thread i we have to show

$$I.pa \notin A \cup A_{pt}.$$

As in the proof of $oinv1$ we execute the step of thread i and steps of thread j up to instruction I and get configurations c' and c'' respectively, where

$$A \subseteq c''.\mathcal{O}_{[i]} \quad \text{and} \quad A_{pt} \subseteq c''.pt_{[i]}.$$

Instruction I in thread j still has to be safe, which implies

$$I.pa \notin c''.\mathcal{O}_{[i]} \cup c''.pt_{[i]}.$$

and concludes the proof.

– $oinv3$. Let $I = sb_{[j]}[k]$ be a read instruction in the suspended part of store buffer j, such that

$$I.pa \in \delta^{k}_{sb}(c'_{sbh}, j).ro.$$

Reusing the proof of $oinv3$ from Lemma 1 we get

$$I.pa \in \delta_{sb}^{k}(c_{sbh}, j).ro.$$

To maintain the invariant it is left to show that

$$I.pa \notin A \cup A_{pt}.$$

We have already shown this in the proof for $oinv1$.

– $oinv4$. We need to show:

$$\forall j \neq i.\ A \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset.$$

From the safety condition for RMW we have:

$$\forall j \neq i.\ A \cap (c.\mathcal{O}_{[j]} \cup c.rls_{l[j]} \cup c.rls_{s[j]}) = \emptyset.$$

From the semantics of SB steps and the coupling relation we can conclude:

$$\mathcal{O}_{[j]} \cup acq(exec(sb_{[j]})) \subseteq c.\mathcal{O}_{[j]} \cup c.rls_{l[j]} \cup c.rls_{s[j]}.$$

Thus, we can conclude:

$$\forall j \neq i.\ A \cap (\mathcal{O}_{[j]} \cup acq(exec(sb_{[j]}))) = \emptyset.$$

It is left to show

$$\forall j \neq i.\ A \cap (acq(susp(sb_{[j]}))) = \emptyset.$$

We prove this by contradiction. Assume

$$\exists a \in A.\ \exists j \neq i.\ \exists k \geq |exec(sb_{[j]})|.\ I = sb_{[j]}[k] \wedge (G(I) \vee vW(I)) \wedge a \in I.A.$$

Let n be the number of instructions up to (and including) instruction k in the suspended part of store buffer j:

$$n = k - |exec(sb_{[j]})| + 1.$$

We execute n steps of thread j in the abstract machine. The choice of the steps to be executed (program or memory) depends on the type of the store buffer instructions under consideration. We refer to the resulting configuration as c'':

$$c \overset{\mathrm{p,m}\ n}{\Longrightarrow}_{j} c''.$$

All these instructions can be executed in the abstract machine for the same arguments as in the proof of $oinv1$. After we execute instruction I, we get

$$a \in c''.\mathcal{O}_{[j]}.$$

Configuration c'' is safe. Hence, the RMW step of thread i still has to be safe, which implies

$$\forall j \neq i.\ a \notin c''.\mathcal{O}_{[j]}$$

and gives a contradiction.

- $sinv1$. The proof is identical to the proof of $sinv1$ from Lemma 1 if one uses $safe\text{-}reach_d(c)$ instead of $sinv4(c_{sbh})$ to conclude

$$R \subseteq \mathcal{O}_{[i]} \quad \text{and} \quad R_{pt} \subseteq pt_{[i]}.$$

- $sinv2$. The proof is identical to the proof of $sinv2$ from Lemma 1 if one uses $safe\text{-}reach_d(c)$ instead of $sinv4(c_{sbh})$ to conclude the safety properties of the ownership transfer.
- $sinv3$. From the coupling invariant and $safe\text{-}reach_d(c)$ we have

$$R \subseteq \mathcal{O}_{[i]}.$$

  With $oinv4$ we get

$$\forall j \neq i.\ R \cap \mathcal{O}_{[j]} = \emptyset,$$

  which implies

$$\forall j.\ \mathcal{O}'_{[j]} \cap ro' = \emptyset.$$

  The rest of the proof is identical to the proof of $sinv3$ from Lemma 1.
- $sinv4$. Let $I = sb_{[j]}[k]$ be a ghost instruction or a volatile write in store buffer j. In the proof of $oinv4$ we have already shown that

$$I.A \cap A = \emptyset.$$

  Later in the proof of $pinv1$ we also show

$$I.A \cap A_{pt} = \emptyset.$$

  The rest of the proof is identical to the proof of $sinv4$ from Lemma 1 just use $safe\text{-}reach_d(c)$ instead of $sinv4(c_{sbh})$.
- $sinv5$. Let $I = sb_{[j]}[k]$ be a write in store buffer j. We can reuse the proof of $sinv5$ from Lemma 1 if we show

$$I.pa \notin R.$$

  From safety of the RMW step and from the coupling relation we get $R \subseteq \mathcal{O}_{[i]}$. With identical proof of $sinv5$ in Lemma 1 we conclude the proof.
- $tinv2$ and $tinv3$. Invariant $hinv3(c_{sbh})$ guarantees that all read temporaries in SBs are present in $\mathrm{dom}(c_{sbh}.\vartheta)$. The invariants are now trivially maintained with $tinv1(c_{sbh})$, $tinv2(c_{sbh})$ and $tinv3(c_{sbh})$.
- $dinv2$. The property is obviously maintained since for all $k \leq |is'_{[i]}|$ it holds

$$dom(c_{sbh}.\vartheta) \cup load_t(is_{[i]}[0:k]) = dom(c'_{sbh}.\vartheta) \cup load_t(is'_{[i]}[0:k-1])$$

- $hinv1$. Let $I = sb_{[j]}[k]$ be a read in the suspended part of store buffer j. We can reuse the proof of $hinv1$ from Lemma 1 if we show that addresses of instruction I and of the RMW instruction of thread i are distinct:

$$I.pa \neq pa.$$

From $oinv1(c_{sbh})$ we have

$$I.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \cup \delta^k_{sb}(c_{sbh}, j).ro.$$

We split cases

- $I.pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]}$. Let n be the number of instructions up to instruction k in the suspended part of store buffer j:

  $$n = k - |exec(sb_{[j]})|.$$

  We execute n steps of thread j in the abstract machine starting from configuration c. The choice of the steps to be executed (program or memory) depends on the type of the store buffer instructions under consideration. We refer to the resulting configuration as c'':

  $$c \overset{\mathrm{p,m}\,n}{\Longrightarrow}_j c''.$$

  All these instructions can be executed in the abstract machine for the same arguments as in the proof of $oinv1$. We get

  $$c''.\mathcal{O}_{[j]} = \delta^k_{sb}(c, i).\mathcal{O}_{[j]}.$$

  Configuration c'' is safe. Hence, the RMW step of thread i still has to be safe, which implies

  $$pa \notin c''.\mathcal{O}_{[j]}.$$

- $I.pa \in \delta^k_{sb}(c_{sbh}, j).ro$. From $oinv3(c_{sbh})$ we know that there are no acquires of $I.pa$ in the executed parts of SBs of other threads. Hence,

  $$I.pa \in \delta^k_{sb}(\Delta^{exec}_{sb[\neq j]}(c_{sbh}), j).ro$$

  If we take $n = k - |exec(sb_{[j]})|$ we can rewrite this as

  $$I.pa \in \delta^n_{sb}(\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq j]}(c_{sbh}), j), j).ro$$

  Applying Lemma 6 we get

  $$I.pa \in \delta^n_{sb}(\Delta^{exec}_{sb}(c_{sbh}), j).ro.$$

  As in the previous case, we execute n instructions of thread j in the abstract machine starting from configuration c and get configuration c''. From the coupling invariant and semantics of the abstract machine it follows

  $$c''.ro = \delta^n_{sb}(\Delta^{exec}_{sb}(c_{sbh}), j).ro.$$

  From the safety of the RMW step in configuration c'' we get

  $$pa \notin c''.ro,$$

  which concludes the proof.

- $hinv3$ is trivially maintained with $tinv3(c_{sbh})$.
- $pinv1$. To maintain the invariant it is enough to show for all threads $j \neq i$:

$$A_{pt} \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset.$$

From the semantics of SB steps and the coupling relation we can conclude:

$$pt_{[j]} \cup acq_{pt}(exec(sb_{[j]})) \subseteq c.pt_{[j]} \cup c.rls_{pt_{[j]}}.$$

Thus, we can conclude from the safety of RMW:

$$A_{pt} \cap (pt_{[j]} \cup acq_{pt}(exec(sb_{[j]}))) = \emptyset.$$

It is left to show

$$A_{pt} \cap (acq_{pt}(susp(sb_{[j]}))) = \emptyset.$$

We prove this by contradiction. Assume

$$\exists a \in A_{pt}.\ \exists k \geq |exec(sb_{[j]})|.\ I = sb_{[j]}[k] \wedge (G(I) \vee vW(I)) \wedge a \in I.A_{pt}.$$

As in the proof of $oinv4$ we execute instructions of thread j starting from configuration c until we executed instruction k. The resulting configuration c'' is safe and it holds

$$a \in c''.pt_{[j]}.$$

From the safety of the RMW step in c'' we derive

$$a \notin c''.pt_{[j]}$$

and get a contradiction.

- $pinv2$. To maintain the invariant it is enough to show for all threads $j \neq i$:

$$A \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset.$$

The proof of that is completely analogous to the proof of $pinv1$.

- $pinv3$. From the safety of the RMW step, the coupling relation and $pinv4(c_{sbh})$ we have

$$pt_{[i]} \cap R = \emptyset.$$

Hence, we can conclude the proof as we do in the proof of $pinv3$ from Lemma 1 by replacing $sinv4(c_{sbh})$ with $safe\text{-}reach_d(c)$.

- $pinv4$. The proof trivially follows from the safety of the RMW step.

□

Read, Write and Ghost

Lemma 11 (invariants maintained by G and vW).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \overset{m}{\Longrightarrow}_i c'_{sbh} \wedge$$
$$hd(is_{[i]}) = GHOST\ (A, L, R, W, A_{pt}, R_{pt}) \vee$$
$$hd(is_{[i]}) = \textbf{Write}\ \text{True a (D, f) r (A, L, R, W, A}_{\text{pt}}\text{, R}_{\text{pt}}) \rightarrow inv(c'_{sbh})$$

Proof. Let c' be the configuration of the abstract machine before executing instruction $hd(is_{[i]})$. If the suspended part of SB i is empty, then $c' = c$. Otherwise, to get c' we execute all instructions of thread i from the suspended part of the SB:

$$n = |susp(sb_{[i]})| \quad \text{and} \quad c \overset{\mathrm{m}}{\Longrightarrow}{}_i^n c'.$$

All these instructions can be executed in the abstract machine, because from $minv1$ and the coupling invariant we have for all $I \in sb_{[i]}$:

$$(R(I) \vee W(I)) \rightarrow I.pa \in \mathrm{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).$$

and there are no instructions recorded in SBs, which affect the MMU state or the translation mode. Since c' is a configuration reachable from c it follows

$$safe\text{-}reach_d(c').$$

We now consider invariants which might get broken by the step:

– $oinv2$. First, we show that this property is maintained for volatile writes of threads $j \neq i$. Let $I = sb_{[j]}[k]$ be a volatile write in store buffer j. We have to show

$$I.pa \notin A \cup A_{pt}.$$

The proof of that is identical to the proof of $oinv2$ from Lemma 10 if one starts to execute steps of abstract machine from configuration c', where all instructions from the suspended part of the SB are already executed.
Second, we maintain the property for thread i. If $vW(hd(is[i]))$ we also have to show that the property holds for the new volatile write added to the SB. For all threads $j \neq i$ it must hold

$$pa \notin \mathcal{O}_{[j]} \cup acq(sb_{[j]}) \cup pt_{[j]} \cup acq_{pt}(sb_{[j]}).$$

From the semantics of SB steps and the coupling relation we can conclude:

$$pt_{[j]} \cup acq_{pt}(exec(sb_{[j]})) \subseteq c'.pt_{[j]} \cup c'.rls_{pt_{[j]}}$$
$$\mathcal{O}_{[j]} \cup acq(exec(sb_{[j]})) \subseteq c'.\mathcal{O}_{[j]} \cup c'.rls_{l[j]} \cup c'.rls_{s[j]}.$$

From the safety of the volatile write in configuration c' we conclude

$$pa \notin pt_{[j]} \cup acq_{pt}(exec(sb_{[j]})) \cup \mathcal{O}_{[j]} \cup acq(exec(sb_{[j]})).$$

It is left to show

$$pa \notin acq_{pt}(susp(sb_{[j]})) \cup \mathcal{O}_{[j]} \cup acq(susp(sb_{[j]})).$$

We show this by contradiction. Let $I = susp(sb_{[j]})[k]$ be an instruction in the suspended part of store buffer j such that $pa \in I.A \cup I.A_{pt}$. We execute $k+1$ program/memory steps of thread j starting from configuration c'. The choice of the steps to be executed depends on the type of the store buffer

instructions under consideration. We refer to the resulting configuration as c'':

$$c' \overset{\text{p,m}}{\Longrightarrow}{}^{k+1}_{j} c''.$$

It holds

$$I.A \subseteq c''.\mathcal{O}_{[j]} \quad \text{and} \quad I.A_{pt} \subseteq c''.pt_{[j]}.$$

Configuration c'' is safe. Hence, the memory step of thread i still has to be safe, which implies

$$pa \notin c''.\mathcal{O}_{[j]} \cup c''.pt_{[j]}$$

and gives a contradiction.

- *oinv*3. The proof is completely analogous to the proof of *oinv*3 from Lemma 10 if one starts executing steps from configuration c', where all instructions from the suspended part of SB i are already executed.
- *oinv*4. The proof is completely analogous to the proof of *oinv*4 from Lemma 10 if one considers c' as the initial configuration of the abstract machine.
- *sinv*4. The property is trivially maintained for old instructions in SBs. For the newly added instruction we conclude from the safety condition of c':

$$\begin{aligned}
&A \subseteq c'.shared \cup c'.\mathcal{O}_{[i]} \cup R_{pt} \setminus A_{pt} \wedge L \subseteq A \wedge \\
&A \cap R = \emptyset \wedge R \subseteq c'.\mathcal{O}_{[i]} \wedge A_{pt} \cap R_{pt} = \emptyset \wedge \\
&A_{pt} \subseteq c'.shared \cup c'.pt_{[i]} \cup R \setminus A \wedge R_{pt} \subseteq c'.pt_{[i]}.
\end{aligned}$$

From the construction of c' and the coupling invariant we have:

$$\begin{aligned}
c'.shared &= \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared \\
c'.\mathcal{O}_{[i]} &= \Delta_{sb}(c_{sbh}, i).\mathcal{O}_{[i]} \\
c'.pt_{[i]} &= \Delta_{sb}(c_{sbh}, i).pt_{[i]}.
\end{aligned}$$

To get the desired property it is left to show that:

$$\forall a \in A_{pt} \cup A.\ a \in c'.shared \rightarrow a \in \Delta_{sb}(c_{sbh}, i).shared. \tag{7}$$

From Lemma 6 we get

$$\Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).shared.$$

From the definition of Δ_{sb}^{exec} and Δ_{sb} we can conclude:

$$\begin{aligned}
&\Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).shared \subseteq \\
&\Delta_{sb}(c_{sbh}, i).shared \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]})))
\end{aligned}$$

From the construction of c' and the coupling relation we have:

$$\begin{aligned}
rels(exec(sb_{[j]})) &\subseteq c'.rls_{s[j]} \cup c'.rls_{l[j]} \\
rels_{pt}(exec(sb_{[j]})) &\subseteq c'.ts_{[j]}.rls_{pt}.
\end{aligned}$$

From the safety condition of c' we have:

$$(A \cup A_{pt}) \cap \bigcup_{\forall j \neq i} (c'.rls_{s[j]} \cup c'.rls_{l[j]} \cup c'.ts_{[j]}.rls_{pt}) = \emptyset,$$

which implies (7).

– *sinv*5. We only consider thread i. We have to show that:

$$pa \notin \Delta_{sb}(c_{sbh}, i).ro. \tag{8}$$

With $safe\text{-}reach_d(c')$, we can conclude:

$$pa \notin c'.ro.$$

From the construction of c' and the coupling relation, we can get:

$$c'.ro = \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), j).ro$$

From Lemma 6 we have:

$$\Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), j).ro = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, j)).ro.$$

From the semantics, we have:

$$\Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, j)).ro \supseteq$$
$$\Delta_{sb}(c_{sbh}, j).ro \setminus (\bigcup_{\forall j \neq i} acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))).$$

With $oinv2(c_{sbh})$, we can get

$$pa \notin \bigcup_{\forall j \neq i} acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))$$

and concludes (8).

– *hinv*4 is trivially maintained with *dinv*2.

– *hinv*6. From $hinv6(c_{sbh})$ we have for all $k < |sb_{[i]}|$:

$$\exists is.\ ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} = is \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]).$$

From the semantics of the SB machine we get for all prefixes is:

$$ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) \circ is'_{[i]} = ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]}$$
$$is \circ p\text{-}ins(sb'_{[i]}[k : |sb'_{[i]}| - 1]) = is \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]).$$

For $k = |sb_{[i]}|$ we get

$$ins(sb'_{[i]}[k]) \circ is'_{[i]} = is_{[i]}$$
$$p\text{-}ins(sb_{[i]}[k]) = [].$$

Hence, the invariant is maintained.

– *pinv*1 and *pinv*2. The proof of these invariants is completely analagous to the proof of *pinv*1 and *pinv*2 from Lemma 10 if one considers c' as the initial configuration of the abstract machine.

□

Lemma 12 (invariants maintained by nvW).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \stackrel{m}{\Longrightarrow}_i c'_{sbh} \wedge$$
$$hd(is_{[i]}) = \textbf{Write}\ \text{False a (D, f) r annot} \rightarrow inv(c'_{sbh})$$

Proof. We first obtain configuration c', where all suspended instruction of thread i are executed, the same way as we do in lemma 11. We now consider invariants which might get broken by the step:

– *oinv*1. Following the proof in Lemma 11, we can choose same translated address *pa* of *a* corresponds to that of store buffer machine. From the safety of configuration c' we have

$$pa \in c'.\mathcal{O}_{[i]}.$$

From the construction of c' and the coupling invariant we have:

$$c'.\mathcal{O}_{[i]} = \Delta_{sb}(c_{sbh}, i).\mathcal{O}_{[i]}.$$

Hence, the desired property for the instruction added to the SB holds.

– *sinv*1. Let *pa* be the translated address of *a*. From the safety of configuration c' we have

$$pa \notin c'.shared.$$

From the construction of c' and the coupling invariant we have:

$$c'.shared = \Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared$$

From Lemma 6 we get

$$\Delta_{sb}(\Delta_{sb}^{exec}(c_{sbh}), i).shared = \Delta_{sb}^{exec}(\Delta_{sb}(c_{sbh}, i)).shared.$$

We conclude the proof with $oinv1(c'_{sbh})$, $oinv4(c_{sbh})$ and $pinv2(c_{sbh})$.

– *sinv*5. The proof follows immediately from $sinv1(c'_{sbh})$ and Lemma 1.
– *hinv*4 is trivially maintained with *dinv*2.
– *hinv*6. The proof is identical to the proof of *hinv*6 in Lemma 11.

□

Lemma 13 (vR implies no vW in SB).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \wedge$$
$$hd(is_{[i]}) = \textbf{Read}\ \text{True a t r} \rightarrow susp(sb_{[i]}) = []$$

Proof. From the coupling invariant for the dirty flag we have

$$c.\mathcal{D}_{[i]} \vee \exists I \in sb_{[i]}.\ vW(I) = \mathcal{D}_{[i]}.$$

We prove by contradiction. Assume

$$\exists I \in sb_{[i]}.\ vW(I).$$

We obtain configuration c', where all suspended instruction of thread i are executed, the same way as we do in lemma 11. Since there is a volatile write I in the suspended part of the store buffer we can conclude

$$c'.\mathcal{D}_{[i]}.$$

But this contradicts to the safety of the volatile read in configuration c'. □

Lemma 14 (invariants maintained by R).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \stackrel{m}{\Longrightarrow}_i c'_{sbh} \wedge$$
$$hd(is_{[i]}) = \textbf{Read}\ \text{vol a t r} \rightarrow inv(c'_{sbh})$$

Proof. We first obtain configuration c', where all suspended instruction of thread i are executed, the same way as we do in lemma 11.

We now consider invariants which might get broken by the step:

- $oinv1$. If $vR(hd(is_{[i]}))$ the invariant is trivially maintained. Otherwise we need to show for the newly added non-volatile read:

  $$pa \in \Delta_{sb}(c_{sbh}, i).\mathcal{O}_{[i]} \cup \Delta_{sb}(c_{sbh}, i).ro.$$

  From the safety of configuration c' we have

  $$pa \in c'.\mathcal{O}_{[i]} \cup c'.ro.$$

  From the construction of c' and the coupling invariant we have:

  $$c'.\mathcal{O}_{[i]} = \Delta_{sb}(c_{sbh}, i).\mathcal{O}_{[i]}$$
  $$c'.ro = \Delta_{sb}(\Delta^{exec}_{sb}(c_{sbh}), i).ro.$$

  To get the desired property it is left to show that:

  $$pa \in c'.ro \rightarrow pa \in \Delta_{sb}(c_{sbh}, i).ro \tag{9}$$

  From Lemma 6 we get

  $$\Delta_{sb}(\Delta^{exec}_{sb}(c_{sbh}), i).ro = \Delta^{exec}_{sb}(\Delta_{sb}(c_{sbh}, i)).ro.$$

  From the definition of Δ^{exec}_{sb} and Δ_{sb} we can conclude:

  $$\Delta^{exec}_{sb}(\Delta_{sb}(c_{sbh}, i)).ro \subseteq$$
  $$\Delta_{sb}(c_{sbh}, i).ro \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]})))$$

From the construction of c' and the coupling relation we have:

$$\begin{aligned} rels(exec(sb_{[j]})) &\subseteq c'.rls_{s[j]} \cup c'.rls_{l[j]} \\ rels_{pt}(exec(sb_{[j]})) &\subseteq c'.rls_{pt[j]}. \end{aligned}$$

From the safety condition of c' we have:

$$pa \notin \bigcup_{\forall j \neq i} (c'.rls_{s[j]} \cup c'.rls_{l[j]} \cup c'.rls_{pt[j]}).$$

which implies (9).

– *oinv*3. For the case of volatile read there is nothing to show because of Lemma 13. For a non-volatile read we need to show for all $j \neq i$:

$$pa \in \Delta_{sb}(c'_{sbh}, i).ro \rightarrow pa \notin (\mathcal{O}'_{[j]} \cup acq(sb'_{[j]}) \cup pt'_{[j]} \cup acq_{pt}(sb'_{[j]})).$$

From the construction of c' and the coupling relation we have

$$\begin{aligned} c'.\mathcal{O}_{[j]} &= \Delta^{exec}_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \\ c'.pt_{[j]} &= \Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \\ c'.ro &= \Delta_{sb}(\Delta^{exec}_{sb}(c_{sbh}), i).ro. \end{aligned}$$

From the safety condition of c' we have:

$$\begin{aligned} &pa \in c'.\mathcal{O}_{[i]} \cup c'.ro \\ &pa \notin \bigcup_{\forall j \neq i} (c'.rls_{s[j]} \cup c'.rls_{l[j]} \cup c'.ts_{[j]}.rls_{pt}). \end{aligned}$$

Applying lemmas 9 and 8 we get

$$pa \notin c'.\mathcal{O}_{[j]} \cup c'.pt_{[j]}.$$

From the semantics of SB steps and the coupling relation we can conclude:

$$\begin{aligned} \mathcal{O}'_{[j]} \cup acq(exec(sb'_{[j]})) &\subseteq c'.\mathcal{O}_{[j]} \cup c'.rls_{l[j]} \cup c'.rls_{s[j]} \\ pt'_{[j]} \cup acq_{pt}(exec(sb'_{[j]})) &\subseteq c'.pt_{[j]} \cup c'.rls_{pt[j]}. \end{aligned}$$

Hence, all it is left to show

$$pa \notin acq(susp(sb'_{[j]})) \cup acq_{pt}(susp(sb'_{[j]})).$$

We have already done this kind of proof for invariant *oinv*2 in Lemma 11.

– *tinv*2 and *tinv*3. Invariant $hinv3(c_{sbh})$ guarantees that all read temporaries in SBs are present in $dom(\vartheta_{[i]})$. The invariants are now trivially maintained with $tinv1(c_{sbh})$, $tinv2(c_{sbh})$ and $tinv3(c_{sbh})$.

– *dinv*2. The property is obviously maintained since for all $k \leq |is'_{[i]}|$ it holds

$$dom(c_{sbh}.\vartheta) \cup load_t(is_{[i]}[0:k]) = dom(c'_{sbh}.\vartheta) \cup load_t(is'_{[i]}[0:k-1])$$

- $hinv1$. For the case of volatile read there is nothing to show because of Lemma 13. For a non-volatile read I added to the store buffer as a result of the step we obviously get

$$I.v = fwd(sb_{[i]}, m, I.pa) = \delta_{sb}^{k}(c_{sbh}, i).m(I.pa).$$

- $hinv2$. Invariant is easily maintained with Lemma 13.
- $hinv3$. For the newly added read the property is trivially maintained. For old reads in the SB the property follows from $tinv3(c_{sbh})$.
- $hinv4$. Invariant is trivially maintained with $tinv3(c_{sbh})$.
- $hinv6$. The proof is identical to the proof of $hinv6$ in Lemma 11.

□

6.5 MMU and PF Steps

Lemma 15 (invariants maintained by MMU).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \stackrel{mu}{\Longrightarrow} c'_{sbh} \rightarrow inv(c'_{sbh})$$

Proof. The only invariants which might get broken by MMU steps are $hinv1$ and $minv1$.

- $hinv1(c_{sbh})$. This invariants can only get broken by the MMU write step. Let pa be the address written by the MMU. From $hinv1(c_{sbh})$ we have

$$\forall k < |sb_{[j]}|.\ k \geq |exec(sb_{[j]})| \wedge nvR(I) \rightarrow I.v = \delta_{sb}^{k}(c_{sbh}, j).m(I.pa),$$

  where $I = sb_{[j]}[k]$. To maintain the invariant we have to show

$$\forall k < |sb_{[j]}|.\ k \geq |exec(sb_{[j]})| \wedge nvR(I) \rightarrow I.pa \neq pa.$$

  For case $j \neq i$ we can reuse the proof for $hinv1$ from Lemma 10. For case $j = i$ we prove this lemma by contradiction. We assume:

$$\exists k < |sb_{[i]}|.\ k \geq |exec(sb_{[i]})| \wedge I = sb_{[i]}[k] \wedge nvR(I) \wedge I.pa = pa.$$

  Let n be the number of instructions up to instruction k in the suspended part of store buffer i:

$$n = k - |exec(sb_{[i]})|$$

  We execute n program/memory steps of thread i in the abstract machine starting from configuration c. The choice of the steps to be executed depends on the type of the store buffer instructions under consideration. We refer to the resulting configuration as c':

$$c \stackrel{\mathrm{p,m}}{\Longrightarrow}{}_{i}^{n} c'.$$

  Since we don't do any MMU steps in this execution, it holds

$$c'.mmu_{[i]} = c.mmu_{[i]} = mmu_{[i]}.$$

Hence, we can execute the MMU write to address pa in configuration c' and this write has to be safe, because configuration c' is safe. This implies

$$pa \notin c'.ro \cup c'.\mathcal{O}_{[i]}.$$

At the same time, instruction I, which is at the head of the instruction list of thread i in configuration c', also has to be safe, which implies:

$$pa \in c'.\mathcal{O}_{[i]} \cup c'.ro$$

and gives a contradiction.

– $minv1$. We conclude the proof with the monotonicity property for MMU reads.

□

Lemma 16 (invariants maintained by page fault).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \xRightarrow{pf} c'_{sbh} \rightarrow inv(c'_{sbh})$$

Proof. The only invariant which might get broken is

– $hinv6$. Because the store buffer and the instruction sequence are both flushed after the page fault step, the invariant is trivially maintained.

□

7 Proving Simulation

In this section we prove simulation between the SB and the virtual machines.

7.1 SB Steps

Lemma 17 (coupling maintained when R, nvW, G exits SB).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge hd(sb_{[i]}) = I \wedge \neg vW(I) \wedge$$
$$c'_{sbh} = \delta_{sb}(c_{sbh}, i) \rightarrow c'_{sbh} \sim c$$

Proof. Since the suspended part of SB i is unchanged, the coupling for the instruction sequence, program state and temporaries is trivially maintained. The coupling for the dirty flag, translation mode and MMU state also can not be broken.

For the other parts of the coupling relation we first observe that

$$\Delta^{exec}_{sb}(c_{sbh}, i) = \Delta^{exec}_{sb}(c'_{sbh}, i).$$

Hence, for $X' \in \{\mathcal{O}, pt, rls_l, rls_s, rls_{pt}\}$ we trivially get

$$c.X'_{[i]} = \Delta^{exec}_{sb}(c_{sbh}, i).X'_{[i]} = \Delta^{exec}_{sb}(c'_{sbh}, i).X'_{[i]}.$$

$$
\begin{aligned}
&k \leq |exec(sb_{[i]})| \wedge \forall X \in \{shared, ro, m\}.\ c.X = \delta_{sb}^{k}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i).X \wedge \\
&\forall j.\ c.mode_{[j]} = mode_{[j]} \wedge c.mmu_{[j]} = mmu_{[j]} \wedge \\
&\quad (c.\mathcal{D}_{[j]} \vee \exists I \in sb_{[j]}.\ vW(I) \leftrightarrow \mathcal{D}_{[j]}) \ \wedge \forall X \in \{\mathcal{O}, pt, rls_l, rls_s, rls_{pt}\}. \\
&\quad (j \neq i \rightarrow c.X_{[j]} = \Delta_{sb}^{exec}(c_{sbh}, j).X_{[j]} \wedge \\
&\qquad\qquad c.is_{[j]} \circ p\text{-}ins(susp(sb_{[j]})) = ins(susp(sb_{[j]})) \circ is_{[j]} \wedge \\
&\qquad\qquad c.\vartheta_{[j]} = del\text{-}t(\vartheta_{[j]}, susp(sb_{[j]})) \wedge c.p_{[j]} = hd\text{-}p(p_{[j]}, susp(sb_{[j]}))) \wedge \\
&\quad (j = i \rightarrow c.ts_{[j]}.X = \Delta_{sb}^{k}(c_{sbh}, j).X \wedge \\
&\qquad\qquad c.is_{[j]} \circ p\text{-}ins(sb_{[j]}[k : |sb_{[j]}| - 1]) = ins(sb_{[j]}[k : |sb_{[j]}| - 1]) \circ is_{[j]} \wedge \\
&\qquad\qquad c.\vartheta_{[j]} = del\text{-}t(\vartheta_{[j]}, sb_{[j]}[k : |sb_{[j]}| - 1]) \wedge \\
&\qquad\qquad c.p_{[j]} = hd\text{-}p(p_{[j]}, sb_{[j]}[k : |sb_{[j]}| - 1])).
\end{aligned}
$$

Fig. 13: Definition of the intermediate coupling relation $sim(c, c_{sbh}, i, k)$.

With Lemma 1 we get $inv(c'_{sbh})$. For $X \in \{shared, ro, m\}$ we conclude

$$
\begin{aligned}
c.X &= \Delta_{sb}^{exec}(c_{sbh}).X && \text{(coupling)} \\
&= \Delta_{sb[\neq i]}^{exec}(\Delta_{sb}^{exec}(c_{sbh}, i)).X && \text{(lemma 6)} \\
&= \Delta_{sb[\neq i]}^{exec}(\Delta_{sb}^{exec}(c'_{sbh}, i)).X \\
&= \Delta_{sb}^{exec}(c'_{sbh}).X. && \text{(lemma 6)}
\end{aligned}
$$

□

When a volatile write exits SB, the virtual machine executes not only this volatile write, but also all local instructions recorded in the SB after the volatile write. To show that the coupling invariant is maintained after all these steps, we define intermediate coupling relation

$$sim(c, c_{sbh}, i, k),$$

which has to hold after k local instructions of thread i are executed in the virtual machine (Fig. 13). In case $k = 0$, relation $sim(c, c_{sbh}, i, 0)$ couples the states after the volatile write is committed to the memory in the SB machine and after the volatile write instruction is executed in the virtual machine. After we execute all local steps before the next volatile write and have $k = |exec(c_{sbh}.sb_{[i]})|$ it holds

$$sim(c, c_{sbh}, i, k) \equiv c_{sbh} \sim c.$$

Lemma 18 (instruction list not empty).

$$
\begin{aligned}
&c.is_{[i]} \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]) = ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} \wedge \\
&k < |sb_{[i]}| \wedge hinv6(c_{sbh}) \wedge \neg P(sb_{[i]}[k]) \rightarrow c.is_{[i]} \neq []
\end{aligned}
$$

Proof. If $k = |sb_{[i]}| - 1$ then we get

$$c.is_{[i]} = ins(sb_{[i]}[k]) \circ is_{[i]}$$

and the lemma trivially holds. Otherwise we prove by contradiction. Let $I = sb_{[i]}[k]$ and $c.is_{[i]} = []$. Then we conclude

$$\begin{aligned} p\text{-}ins(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) &= p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \\ &= ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \circ is_{[i]} \\ &= ins(I) \circ ins(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \circ is_{[i]}. \end{aligned}$$

From $hinv6(c_{sbh})$ we know that there exists is' such that

$$\begin{aligned} &ins(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \circ is_{[i]} \\ &= is' \circ p\text{-}ins(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \\ &= is' \circ ins(I) \circ ins(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \circ is_{[i]}, \end{aligned}$$

which gives a contradiction. □

Lemma 19 (unowned nvW is in local release set).

$$\begin{aligned} \forall j, k, pa.\ & k < |exec(sb_{[j]})| \wedge (c_{sbh} \sim c \vee sim(c, c_{sbh}, i, n) \wedge i \neq j) \wedge inv(c_{sbh}) \wedge \\ & nvW(sb_{[j]}[k]) \wedge pa = sb_{[j]}[k].pa \wedge (pa \notin c.\mathcal{O}_{[j]} \vee pa \in c.shared) \rightarrow \\ & pa \in c.rls_{l[j]} \end{aligned}$$

Proof. From $oinv1(c_{sbh})$ and $sinv1(c_{sbh})$ we have:

$$pa \in \delta^k_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \wedge pa \notin \delta^k_{sb}(c_{sbh}, j).shared.$$

The coupling relations for thread j gives us

$$pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \vee pa \in \Delta^{exec}_{sb}(c_{sbh}).shared.$$

or

$$pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \vee pa \in \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared$$

depending of what kind of simulation relation holds. From $oinv4(c_{sbh})$ and $pinv2(c_{sbh})$ it follows for all threads $l \neq j$:

$$pa \notin \mathcal{O}[l] \cup acq(sb_{[l]}) \cup pt[l] \cup acq_{sb}(sb_{[l]}).$$

Hence, with $sinv4(c_{sbh})$ we can conclude that pa is not released by any instruction in the executed part of thread l. Thus, we get

$$pa \notin \Delta^{exec}_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \vee pa \in \Delta^{exec}_{sb}(c_{sbh}, j).shared.$$

- Let $pa \notin \Delta_{sb}^{exec}(c_{sbh}, j).\mathcal{O}_{[j]}$. In the rest of the executed part of the store buffer there must be an instruction, which removes pa from the owns set of thread j. When an address is removed from the ownership set, it is added to one of the release sets. In order to be added to the shared release set, the address has to be shared at the time of the ownership transfer. We have already shown that no thread other than j can make pa shared. The only way for thread j to make an owned unshared address shared, is by releasing it, which puts the address to the local release set.
- Let $pa \in \Delta_{sb}^{exec}(c_{sbh}, j).shared$. In this case in the rest of the executed part of the store buffer there has to be an instruction, which releases pa and adds it to the local release set.

Hence, we can conclude

$$pa \in \Delta_{sb}^{exec}(c_{sbh}, j).rls_{l[j]} = c.rls_{l[j]}.$$

□

Lemma 20 (sim implies disjoint sets).

$$\forall i.\ sim(c, c_{sbh}, i, n) \wedge inv(c_{sbh}) \rightarrow disjoint\text{-}osets(c)$$

Proof. For thread i we have to prove:

$$c.\mathcal{O}_{[i]} \cap c.ro = \emptyset$$
$$c.pt_{[i]} \cap c.shared = \emptyset.$$

With the help of the coupling relation this is transformed to

$$\Delta_{sb}^{n}(c_{sbh}, i).\mathcal{O}_{[i]} \cap \delta_{sb}^{n}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i).ro = \emptyset$$
$$\Delta_{sb}^{n}(c_{sbh}, i).pt_{[i]} \cap \delta_{sb}^{n}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i).shared = \emptyset.$$

With Lemma 6 we have:

$$\delta_{sb}^{n}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i).ro = \Delta_{sb[\neq i]}^{exec}(\delta_{sb}^{n}(c_{sbh}, i)).ro$$
$$\delta_{sb}^{n}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i).shared = \Delta_{sb[\neq i]}^{exec}(\delta_{sb}^{n}(c_{sbh}, i)).shared.$$

From the semantics we can conclude:

$$\Delta_{sb[\neq i]}^{exec}(\delta_{sb}^{n}(c_{sbh}, i)).ro \subseteq \delta_{sb}^{n}(c_{sbh}, i).ro \bigcup_{\forall j \neq i} rels(exec(sb_{[j]}))$$
$$\Delta_{sb[\neq i]}^{exec}(\delta_{sb}^{n}(c_{sbh}, i)).shared \subseteq$$
$$\delta_{sb}^{n}(c_{sbh}, i).shared \bigcup_{\forall j \neq i} rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]})).$$

With $sinv3(c_{sbh})$, $pinv3(c_{sbh})$ and Lemma 1 we have:

$$\Delta_{sb}^{n}(c_{sbh}, i).\mathcal{O}_{[i]} \cap \delta_{sb}^{n}(c_{sbh}, i).ro = \emptyset$$
$$\Delta_{sb}^{n}(c_{sbh}, i).pt_{[i]} \cap \delta_{sb}^{n}(c_{sbh}, i).shared = \emptyset.$$

With $oinv4(c_{sbh})$ we can get:

$$\forall j \neq i.\ \Delta^n_{sb}(c_{sbh}, i).\mathcal{O}_{[i]} \cap (\mathcal{O}_{[j]} \cap acq(exec(sb_{[j]}))) = \emptyset.$$

With $sinv4(c_{sbh})$ we can conclude:

$$rels(exec(sb_{[j]})) \subseteq \mathcal{O}_{[j]} \cap acq(exec(sb_{[j]}))$$

We can conclude:

$$\forall j \neq i.\ \Delta^n_{sb}(c_{sbh}, i).\mathcal{O}_{[i]} \cap rels(exec(sb_{[j]})) = \emptyset$$

With $sinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we can also get:

$$\forall j \neq i.\ \Delta^n_{sb}(c_{sbh}, i).pt_{[i]} \cap (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))) = \emptyset.$$

Thus, we can conclude:

$$c.\mathcal{O}_{[i]} \cap c.ro = \emptyset$$
$$c.pt_{[i]} \cap c.shared = \emptyset.$$

For thread $j \neq i$ we have to prove:

$$c.\mathcal{O}_{[j]} \cap c.ro = \emptyset,$$
$$c.pt_{[j]} \cap c.shared = \emptyset.$$

With the help of the coupling relation this is transformed to

$$\Delta^{exec}_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \cap \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).ro = \emptyset$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared = \emptyset.$$

We prove this case by contradiction. Assume

$$\exists a, a'.\ a \in \Delta^{exec}_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \cap \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).ro \wedge$$
$$a' \in \Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap \delta^n_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared.$$

With Lemma 6 we can get:

$$\Delta^{exec}_{sb}(c_{sbh}).ro \subseteq \Delta^{exec}_{sb}(c_{sbh}, j).ro \bigcup_{k \neq j} rels(exec(sb_{[k]}))$$
$$\Delta^{exec}_{sb}(c_{sbh}).shared \subseteq$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).shared \bigcup_{k \neq j} rels(exec(sb_{[k]})) \cup rels_{pt}(exec(sb_{[k]})).$$

With Lemma 1, $sinv3(c_{sbh})$ and $pinv3(c_{sbh})$

$$\Delta^{exec}_{sb}(c_{sbh}, j).\mathcal{O}_{[j]} \cap \Delta^{exec}_{sb}(c_{sbh}, j).ro = \emptyset$$
$$\Delta^{exec}_{sb}(c_{sbh}, j).pt_{[j]} \cap \Delta^{exec}_{sb}(c_{sbh}, j).shared = \emptyset.$$

From $oinv4(c_{sbh})$, $sinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we can conclude:

$$\Delta_{sb}^{exec}(c_{sbh}, j).\mathcal{O}_{[j]} \cap \Delta_{sb}^{exec}(c_{sbh}).ro = \emptyset$$
$$\Delta_{sb}^{exec}(c_{sbh}, j).pt_{[j]} \cap \Delta_{sb}^{exec}(c_{sbh}).shared = \emptyset.$$

With Lemma 6 and $sinv4(c_{sbh})$ we can conclude

$$a \in acq(sb[n : |exec(sb_{[i]})| - 1]) \wedge$$
$$a' \in acq_{pt}(sb[n : |exec(sb_{[i]})| - 1]).$$

This contradicts to $oinv4(c_{sbh})$ and $pinv1(c_{sbh})$. Thus, we can conclude:

$$c.\mathcal{O}_{[j]} \cap c.ro = \emptyset$$
$$c.pt_{[j]} \cap c.shared = \emptyset.$$

The remaining properties follow immediately from Lemma 1, the coupling relation and and invariants $oinv4(c_{sbh})$, $pinv1(c_{sbh})$, $pinv2(c_{sbh})$, $pinv4(c_{sbh})$ and $sinv3(c_{sbh})$. □

Lemma 21 (no nvW to a read address).

$$\forall i, pa.\ (R(hd(c.is_{[i]})) \vee RMW(hd(c.is_{[i]}))) \wedge (c_{sbh} \sim c \vee sim(c, c_{sbh}, i, n)) \wedge$$
$$safe\text{-}reach_d(c) \wedge pa \in (atran(mmu_{[i]}, hd(is_{[i]}).va, mode_{[i]}, hd(is_{[i]}).r)) \wedge$$
$$inv(c_{sbh}) \rightarrow \forall j \neq i.\ \forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]})[k] \wedge sb_{[j]}[k].pa = pa)$$

Proof. By contradiction. Assume

$$\exists j \neq i.\ \exists k < |exec(sb_{[j]})|.\ sb_{[j]}[k] = \mathbf{Write_{sb}}\ \text{Flase va pa (D, f) r annot.}$$

Applying Lemma 19 we get

$$pa \notin c.\mathcal{O}_{[j]} \vee pa \in c.shared \rightarrow c.rls_{l[j]}.$$

From the safety for reads and RMWs we get

$$pa \in c.\mathcal{O}_{[i]} \cup c.shared \cup c.ro \cup c.pt_{[i]}\ \wedge \forall j \neq i.\ pa \notin c.rls_{l[j]}.$$

Hence, we can conclude

$$pa \in c.\mathcal{O}_{[j]} \wedge pa \notin c.shared.$$

Applying Lemma 9 or Lemma 20 we conclude

$$pa \notin c.\mathcal{O}_{[i]} \cup c.pt_{[i]} \cup c.ro.$$

and get a contradiction. □

Lemma 22 (simulating vW exits sb inductive).

$$sim(c, c_{sbh}, i, k) \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge k < |exec(sb_{[i]})| \wedge$$
$$\forall k' < |exec(sb_{[i]})|.\ I = sb_{[i]}[k'] \wedge \neg vR(I) \wedge (R(I) \rightarrow I.v = \delta_{sb}^{k'}(c_{sbh}, i).m(I.pa))$$
$$\rightarrow \exists c'.\ c \Rightarrow_i c' \wedge sim(c', c_{sbh}, i, k+1)$$

Proof. Let $I = sb_{[i]}[k]$. We execute either a memory or a program step of thread i depending on $sb_{[i]}[k]$:

- case $\neg P(I)$. From Lemma 18 we conclude

$$c.is_{[i]} \neq [].$$

  Hence, from the coupling relation we have

$$hd(c.is_{[i]}) = hd(ins(sb_{[i]}[k : |sb_{[i]}| - 1])) = ins(I).$$

  If I is a read or a write instruction, then from $sim(c, c_{sbh}, i, k)$ and from $minv1(c_{sbh})$ we get

$$I.pa \in \text{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).$$

  Hence, we can always execute the instruction from the head of the instruction list and choose the same translated address as we have previously chosen for the corresponding step of the SB machine. Let c' be configuration of the abstract machine after the step:

$$c \overset{\text{m}}{\Longrightarrow}_i c'.$$

  For the instruction sequence after the step we obviously get

$$\begin{aligned} & c'.is_{[i]} \circ p\text{-}ins(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \\ = {} & tl(c.is_{[i]}) \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}| - 1]) \\ = {} & tl(ins(sb_{[i]}[k : |sb_{[i]}| - 1])) \circ is_{[i]} \\ = {} & ins(sb_{[i]}[k+1 : |sb_{[i]}| - 1]) \circ is_{[i]}. \end{aligned}$$

  The coupling for mode, dirty flag, MMU state and program state is trivially maintained, as well as the coupling for threads other than i. For the remaining parts of the coupling relation we do a further case split:
  - $nvW(I)$. In this case only the coupling for memory can get broken. From the semantics we get

$$c'.m(I.pa) = I.f(c.\vartheta_{[i]}).$$

    From $hinv4(c_{sbh})$ we know that

$$I.v = I.f(\vartheta_{[i]}).$$

From $hinv4(c_{sbh})$, $dinv1(c_{sbh})$ and the coupling relation we get

$$\begin{aligned} I.f(\vartheta_{[i]}) &= I.f(del\text{-}t(\vartheta_{[i]}, sb_{[i]}[k : |sb_{[i]}| - 1])) \\ &= I.f(c.\vartheta_{[i]}). \end{aligned}$$

Hence, we have $c'.m(I.pa) = I.v$, which implies

$$c'.m = \delta_{sb}(\delta^k_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i)).m$$

and concludes the proof for the memory coupling.

- $nvR(I)$. In this case the coupling for the temporaries can get broken. From the semantics we get

$$c'.\vartheta_{[i]} = c.\vartheta_{[i]}(I.t \mapsto c.m(I.pa)).$$

From the precondition of the function we know that

$$I.v = \delta^k_{sb}(c_{sbh}, i).m(I.pa).$$

With Lemma 21 we get for all $j \neq i$:

$$\forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]}[k]) \wedge sb_{[j]}[k].pa = I.pa)$$

Hence,

$$\begin{aligned} c.m(pa) &= \delta^k_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).m(pa) \\ &= \delta^k_{sb}(c_{sbh}, i).m(I.pa) \\ &= I.v. \end{aligned}$$

From invariant $hinv3(c_{sbh})$ we have

$$\vartheta_{[i]}(I.t) = I.v.$$

From the semantics and from the coupling relation we now conclude

$$\begin{aligned} c'.\vartheta_{[i]} &= c.\vartheta_{[i]}(I.t \mapsto I.v) \\ &= del\text{-}t(\vartheta_{[i]}, sb_{[i]}[k : |sb_{[i]}| - 1])(I.t \mapsto I.v) \\ &= del\text{-}t(\vartheta_{[i]}, sb_{[i]}[k + 1 : |sb_{[i]}| - 1]). \end{aligned}$$

- $G(I)$. In this case the store buffer machine only accumulates local updates on shared set when computing the release local and release shared set But the abstract machine with delayed release accumulates global updates on shared set while computing the corresponding components. Hence, the coupling relation for release shared and release local set can get broken. We have to prove:

$$I.R \cap \delta^k_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared = I.R \cap \Delta^k_{sb}(c_{sbh}, i).shared.$$

From Lemma 6 and $sinv4(c_{sbh})$, we can get:

$$\delta^k_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared = \Delta^{exec}_{sb[\neq i]}(\delta^k_{sb}(c_{sbh}, i)).shared$$

$$\begin{aligned}
&\Delta^{exec}_{sb[\neq i]}(\delta^k_{sb}(c_{sbh}, i)).shared \\
&\subseteq \delta^k_{sb}(c_{sbh}, i).shared \bigcup_{\forall j \neq i} rels(exec(sb_j)) \cup rels_{pt}(exec(sb_{[j]})), \\
&\Delta^{exec}_{sb[\neq i]}(\delta^k_{sb}(c_{sbh}, i)).shared \\
&\supseteq \delta^k_{sb}(c_{sbh}, i).shared \setminus (\bigcup_{\forall j \neq i} acq(exec(sb_j)) \cup acq_{pt}(exec(sb_{[j]}))).
\end{aligned}$$

With $sinv4(c_{sbh})$ and $oinv4(c_{sbh})$ we can conclude:

$$\begin{aligned}
I.R \cap (\bigcup_{\forall j \neq i} rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))) = \emptyset, \\
I.R \cap (\bigcup_{\forall j \neq i} acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))) = \emptyset.
\end{aligned}$$

Hence, we have:

$$\begin{aligned}
I.R \cap \delta^k_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared \subseteq I.R \cap \Delta^k_{sb}(c_{sbh}, i).shared, \\
I.R \cap \delta^k_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).shared \supseteq I.R \cap \Delta^k_{sb}(c_{sbh}, i).shared,
\end{aligned}$$

which concludes the proof.

– Case $P(I)$. Then we have from the coupling relation

$$\begin{aligned}
c.p_{[i]} &= hd\text{-}p(p_{[i]}, sb_{[i]}[k : |sb_{[i]}| - 1]) = I.p1 \\
c.\vartheta_{[i]} &= del\text{-}t(\vartheta_{[i]}, sb_{[i]}[k : |sb_{[i]}| - 1]).
\end{aligned}$$

Observing that $load_t(sb_{[i]}[k : |sb_{[i]}| - 1]) = load_t(sb_{[i]}[k + 1 : |sb_{[i]}| - 1])$ we get from $hinv5(c_{sbh})$:

$$\begin{aligned}
\delta_p(c.p_{[i]}, c.\vartheta_{[i]}) &= (I.p2, I.is) \\
I.p2 &= hd\text{-}p(p_{[i]}, sb_{[i]}[k + 1 : |sb_{[i]}| - 1]).
\end{aligned}$$

Hence, we execute the program step from configuration c:

$$c \overset{\mathrm{p}}{\Longrightarrow}_i c'.$$

For the program state the coupling is maintained because

$$c'.p_{[i]} = I.p2 = hd\text{-}p(p_{[i]}, sb_{[i]}[k + 1 : |sb_{[i]}| - 1]).$$

For the instruction sequence we get from the semantics of the abstract machine and from the coupling relation:

$$\begin{aligned} & c'.is_{[i]} \circ p\text{-}ins(sb_{[i]}[k+1 : |sb_{[i]}|-1]) \\ = \; & c.is_{[i]} \circ I.is \circ p\text{-}ins(sb_{[i]}[k+1 : |sb_{[i]}|-1]) \\ = \; & c.is_{[i]} \circ p\text{-}ins(sb_{[i]}[k : |sb_{[i]}|-1]) \\ = \; & ins(sb_{[i]}[k : |sb_{[i]}|-1]) \circ is_{[i]} \\ = \; & ins(sb_{[i]}[k+1 : |sb_{[i]}|-1]) \circ is_{[i]}, \end{aligned}$$

which concludes the proof for the coupling relation.

□

Lemma 23 (simulating vW exits sb).

$$\begin{aligned} & c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge I = hd(sb_{[i]}) \wedge vW(I) \wedge \\ & c_{sbh} \overset{sb}{\Longrightarrow}_i c'_{sbh} \rightarrow \exists c'.\ c \Rightarrow_i^* c' \wedge c'_{sbh} \sim c' \end{aligned}$$

Proof. From Lemma 18 we conclude

$$c.is_{[i]} \neq [].$$

Hence, from the coupling relation we have

$$hd(c.is_{[i]}) = ins(I).$$

From the coupling relation and from $minv1(c_{sbh})$ we conclude

$$I.pa \in \text{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r).$$

Hence, we can always execute the volatile write from the head of the instruction list and choose the same translated address as we have previously chosen for the corresponding step of the SB machine. Let c'' be configuration of the abstract machine after the step:

$$c \overset{\text{m}}{\Longrightarrow}_i c''.$$

From the coupling relation and the semantics of abstract and SB machines we get for $X \in \{shared, ro, m\}$:

$$\begin{aligned} c''.X &= \delta_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).X \\ &= \Delta^{exec}_{sb[\neq i]}(c'_{sbh}).X \quad (Lemma6) \end{aligned}$$

For $X \in \{\mathcal{O}, pt, rls_{pt}\}$ we get:

$$c''.X_{[i]} = \delta_{sb}(c_{sbh}, i).X = c'_{sbh}.X_{[i]}$$

For $X \in \{rls_s, rls_l\}$ we have to prove:

$$I.R \cap \Delta^{exec}_{sb[\neq i]}(c_{sbh}).shared = I.R \cap c_{sbh}.shared \tag{10}$$

From the $sinv4(c_{sbh})$ we can conclude:

$$\Delta^{exec}_{sb[\neq i]}(c_{sbh}).shared \subseteq c_{sbh}.shared \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]})))$$

$$\Delta^{exec}_{sb[\neq i]}(c_{sbh}).shared \supseteq c_{sbh}.shared \setminus (\bigcup_{\forall j \neq i} (acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))))$$

With $sinv4(c_{sbh})$, $oinv4(c_{sbh})$ and the definition of acq and acq_{pt}, we can conclude:

$$I.R \cap \bigcup_{\forall j \neq i} (rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))) = \emptyset$$

$$I.R \cap \bigcup_{\forall j \neq i} (acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]}))) = \emptyset$$

which implies (10).
For the instruction sequence we conclude with the help of the coupling relation:

$$\begin{aligned} c''.is_{[i]} \circ p\text{-}ins(sb'_{[i]}) &= tl(c.is_{[i]}) \circ p\text{-}ins(susp(sb_{[i]})) \\ &= tl(ins(susp(sb_{[i]}))) \circ is_{[i]} \\ &= ins(sb'_{[i]}) \circ is'_{[i]}. \end{aligned}$$

For the temporaries and the program state it obviously holds

$$\begin{aligned} c''.\vartheta_{[i]} &= c.\vartheta_{[i]} \\ &= del\text{-}t(\vartheta_{[i]}, sb_{[i]}) \\ &= del\text{-}t(\vartheta'_{[i]}, sb'_{[i]}) \\ c''.p_{[i]} = c.p_{[i]} &= hd\text{-}p(p_{[i]}, sb_{[i]}) \\ &= hd\text{-}p(p'_{[i]}, sb'_{[i]}). \end{aligned}$$

From the coupling relation we get $\mathcal{D}_{[i]}$, which implies $\mathcal{D}'_{[i]}$. From the semantics of the memory step we also get $c'.\mathcal{D}_{[i]}$. This implies

$$c'.\mathcal{D}_{[i]} \vee \exists I \in sb_{[i]}.\ vW(I) \leftrightarrow \mathcal{D}'_{[i]}.$$

Hence, we have $sim(c'', c'_{sbh}, i, 0)$. Let n be the length of the executed part of SB i in configuration c'_{sbh}:

$$n = |exec(sb'_{[i]})|.$$

With $hinv1(c_{sbh})$ we can conclude the consistency of the read value in both machines. From $hinv2(c_{sbh})$ we know that no volatile read instruction exists in $sb_{[i]}$. Then we apply Lemma 22 $n-1$ times and execute steps of thread i accordingly. Finally we get configuration c', such that

$$c'' \Rightarrow^*_i c' \quad \text{and} \quad sim(c', c'_{sbh}, i, n).$$

To get the coupling $c'_{sbh} \sim c'$ from $sim(c', c'_{sbh}, i, n)$ we only have to show

$$\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(c'_{sbh}), i) = \Delta^{exec}_{sb}(c'_{sbh}),$$

which we easily get with Lemma 1 and Lemma 6. □

7.2 Instruction List Coupling

Lemma 24 (coupling for instructions maintained with vW).

$$c_{sbh} \sim c \wedge (c_{sbh} \overset{m}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{p}{\Longrightarrow}_i c'_{sbh}) \wedge susp(sb'_{[i]}) \neq [] \rightarrow$$
$$c.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) = ins(susp(sb'_{[i]})) \circ is'_{[i]}$$

Proof. Let $I = hd(is_{[i]})$. For case $c_{sbh} \overset{\mathrm{m}}{\Longrightarrow}_i c'_{sbh}$ we conclude

$$\begin{aligned} c.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) &= c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]})) && (\text{def. } \overset{\mathrm{m}}{\Longrightarrow}_i) \\ &= ins(susp(sb_{[i]})) \circ is_{[i]} && (\text{coupling}) \\ &= ins(susp(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\mathrm{m}}{\Longrightarrow}_i) \end{aligned}$$

For case $c_{sbh} \overset{\mathrm{p}}{\Longrightarrow}_i c'_{sbh}$ we have

$$\begin{aligned} c.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) &= c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]})) \circ is' && (\text{def. } \overset{\mathrm{p}}{\Longrightarrow}_i) \\ &= ins(susp(sb_{[i]})) \circ is_{[i]} \circ is' && (\text{coupling}) \\ &= ins(susp(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\mathrm{p}}{\Longrightarrow}_i) \end{aligned}$$

□

Lemma 25 (coupling for instructions maintained without vW).

$$c_{sbh} \sim c \wedge c_{sbh} \overset{m}{\Longrightarrow}_i c'_{sbh} \wedge c \overset{m}{\Longrightarrow}_i c' \wedge susp(sb'_{[i]}) = [] \rightarrow$$
$$c'.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) = ins(susp(sb'_{[i]})) \circ is'_{[i]}$$

Proof. Let $I = hd(is_{[i]})$. We conclude

$$\begin{aligned} & c'.is_{[i]} \circ p\text{-}ins(susp(sb'_{[i]})) \\ &= tl(c.is_{[i]}) \circ p\text{-}ins(susp(sb'_{[i]})) && (\text{def. } \overset{\mathrm{m}}{\Longrightarrow}_i) \\ &= tl(c.is_{[i]}) \circ p\text{-}ins(susp(sb_{[i]})) \\ &= tl(c.is_{[i]} \circ p\text{-}ins(susp(sb_{[i]}))) && (\text{def. } tl) \\ &= tl(ins(susp(sb_{[i]})) \circ is_{[i]}) && (\text{coupling}) \\ &= tl(is_{[i]}) && (\text{no vW}) \\ &= ins(susp(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\mathrm{m}}{\Longrightarrow}_i) \end{aligned}$$

□

7.3 Program Step

For a program step we make a case distinction on whether there is an outstanding volatile write in the SB. When there is a volatile write in the SB, the abstract machine does not perform any steps. Otherwise, both machines make the same step.

Lemma 26 (simulating program step with vW).

$$\forall i.\ c_{sbh} \sim c \wedge (\exists k.\ vW(sb_{[i]}[k])) \wedge c_{sbh} \overset{p}{\Longrightarrow}_i c'_{sbh} \rightarrow c'_{sbh} \sim c$$

Proof. From the coupling invariant and the semantics of the program step we have

$$\begin{aligned} c.p_{[i]} &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb_{[i]})) \\ &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb_{[i]}) \circ \mathit{PROG}_{sbh}\ p_{[i]}\ p'_{[i]}\ is') \\ &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb'_{[i]})). \end{aligned}$$

In $\mathit{susp}(sb'_{[i]})$ there is now at least one program instruction. Hence, we have

$$\mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb'_{[i]})) = \mathit{hd\text{-}p}(p'_{[i]}, \mathit{susp}(sb'_{[i]})),$$

and the coupling relation for the program state is maintained.

Coupling for the instruction sequence is maintained with Lemma 24. All the other parts of the coupling invariant are trivially maintained. □

Lemma 27 (simulating program step without vW).

$$\forall i.\ c_{sbh} \sim c \wedge (\forall k.\ \neg vW(sb_{[i]}[k])) \wedge c_{sbh} \overset{p}{\Longrightarrow}_i c'_{sbh} \wedge c \overset{p}{\Longrightarrow}_i c' \rightarrow c'_{sbh} \sim c_{sbh}$$

Proof. Observing that the suspended part of $sb_{[i]}$ is empty, we get from the coupling invariant

$$\begin{aligned} c.p_{[i]} &= \mathit{hd\text{-}p}(p_{[i]}, \mathit{susp}(sb_{[i]})) = p_{[i]} \\ c.\vartheta_{[i]} &= \mathit{del\text{-}t}(\vartheta_{[i]}, \mathit{susp}(sb_{[i]})) = \vartheta_{[i]} \\ c.is_{[i]} &= c.is_{[i]} \circ \mathit{p\text{-}ins}(\mathit{susp}(sb_{[i]})) = is_{[i]}. \end{aligned}$$

Hence, the resulting program state and the generated instruction sequence in both machines are the same:

$$\delta_p(c.p_{[i]}, c.\vartheta_{[i]}) = \delta_p(p_{[i]}, \vartheta_{[i]}) = (p', is').$$

For the coupling of the program state we trivially get

$$c'.p_{[i]} = p'_{[i]} = \mathit{hd\text{-}p}(p'_{[i]}, \mathit{susp}(sb'_{[i]})).$$

Coupling for the instruction sequence we have

$$\begin{aligned} & c'.is_{[i]} \circ \mathit{p\text{-}ins}(\mathit{susp}(sb'_{[i]})) \\ &= c.is_{[i]} \circ is' \circ \mathit{p\text{-}ins}(\mathit{susp}(sb_{[i]})) && (\text{def. } \overset{\mathrm{p}}{\Longrightarrow}_i) \\ &= c.is_{[i]} \circ \mathit{p\text{-}ins}(\mathit{susp}(sb_{[i]})) \circ is' && (\text{no vW}) \\ &= \mathit{ins}(\mathit{susp}(sb_{[i]})) \circ is_{[i]} \circ is' && (\text{coupling}) \\ &= \mathit{ins}(\mathit{susp}(sb'_{[i]})) \circ is'_{[i]}. && (\text{def. } \overset{\mathrm{p}}{\Longrightarrow}_i) \end{aligned}$$

All the other parts of the coupling invariant are trivially maintained. □

7.4 MMU and PF Steps

In case of any MMU step the same action is performed in both machines.

Lemma 28 (no nvW to page tables).

$$\forall i, a.\ c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}mmu\text{-}acc_d(c, a, i) \rightarrow \\ (\forall j, k.\ k < |exec(sb_{[j]})| \wedge nvW(sb_{[j]}[k]) \rightarrow sb_{[j]}[k].pa \neq a)$$

Proof. We prove this lemma by contradiction. Let $I = sb_{[j]}[k]$ and

$$\exists j.\ \exists k < |exec(sb_{[j]})|.\ nvW(I) \wedge I.pa = a.$$

Lemma 19 gives us

$$a \in c.\mathcal{O}_{[j]} \cup c.rls_{l[j]},$$

which contradicts to $safe\text{-}mmu\text{-}acc_d(c, a, i)$. □

As an important consequence of Lemma 28 we get the equality of page table memory content in SB and abstract machines in case the coupling relation holds.

Lemma 29 (simulating MMU and PF steps).

$$\forall i,\ c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge (c_{sbh} \overset{mu}{\Longrightarrow}_i c'_{sbh} \vee c_{sbh} \overset{pf}{\Longrightarrow}_i c'_{sbh}) \rightarrow \\ \exists c'.\ (c \overset{mu}{\Longrightarrow}_i c' \vee c \overset{pf}{\Longrightarrow}_i c') \wedge c'_{sbh} \sim c'$$

Proof. We consider cases depending on the type of the step:

- MMU read from address a. In this case we have

  $$mmu'_{[i]} = \delta_{mmur}(mmu_{[i]}, a, m(a)) \wedge can\text{-}access(mmu_{[i]}, a).$$

  We let c' is the configuration after c performs a mmu read step. From the semantics of the MMU step we have

  $$c'.mmu_{[i]} = \delta_{mmur}(c.mmu_{[i]}, a, c.m(a)).$$

  From the coupling invariant we have $mmu_{[i]} = c.mmu_{[i]}$. Hence, we know that $can\text{-}access(c.mmu_{[i]}, a)$ holds and from the safety of the abstract machine we get

  $$safe\text{-}mmu\text{-}acc_d(c, a, i).$$

  With Lemma 28 we conclude $m(a) = c.m(a)$. Hence,

  $$mmu'_{[i]} = c'.mmu_{[i]}.$$

– MMU write to address a. We have

$$c'_{sbh}.m(a) = x \wedge x \in \delta_{mmuw}(mmu_{[i]}, a, m(a)) \wedge \mathit{can\text{-}access}(mmu_{[i]}, a).$$

As in the previous case, we conclude

$$\mathit{safe\text{-}mmu\text{-}acc}_d(c, a, i).$$

Since $mmu_{[i]} = c.mmu_{[i]}$, we know that $x \in \delta_{mmuw}(c.mmu_{[i]}, a, c.m(a))$. Hence, we perform the same step in the abstract machine and get

$$c'.m(a) = x = c'_{sbh}.m(a).$$

With Lemma 28 we conclude the proof for the memory coupling.

– Page fault at address pa. As in previous case, we can get

$$\mathit{can\text{-}access}(c.mmu_{[i]}, pa) \ \wedge \ \mathit{safe\text{-}mmu\text{-}acc}_d(c, pa, i)$$

With Lemma 28, we can conclude $m(pa) = c.m(pa)$. We have $sb_{[i]} = []$. With the coupling relation, we can conclude:

$$c.is_{[i]} = is_{[i]} \ \wedge \ c.p_{[i]} = p_{[i]}$$

Let $I = hd(is_{[i]}) = hd(c.is_{[i]})$, we have:

$$\mathit{can\text{-}page\text{-}fault}(c.mmu_{[i]}, I.va, I.r, pa, c.m(pa))$$

For mmu state of thread i, we have:

$$\begin{aligned} mmu'_{[i]} &= \delta_{flush}(mmu_{[i]}, \{I.va\}, 0) && \text{(semantics)} \\ &= \delta_{flush}(c.mmu_{[i]}, \{I.va\}, 0) && \text{(coupling)} \\ &= c.mmu'_{[i]} && \text{(semantics)} \end{aligned}$$

For program state of thread i, we have:

$$\begin{aligned} p'_{[i]} &= \delta_{pf}(p_{[i]}, I.va, I.r, m(pa)) && \text{(semantics)} \\ &= \delta_{pf}(c.p_{[i]}, I.va, I.r, c.m(pa)) && \text{(coupling)} \\ &= c'.p_{[i]} && \text{(semantics)} \end{aligned}$$

With $sb'_{[i]} = []$, we can conclude:

$$c'.p_{[i]} = \mathit{hd\text{-}p}(p'_{[i]}, \mathit{susp}(sb'_{[i]}))$$

From the semantics, we also have:

$$is'_{[i]} = c'.is_{[i]} = [] \ \wedge \ rls'_{[i]} = c'.rls_{[i]} = \emptyset \ \wedge \ \neg\mathcal{D}'_{[i]} \ \wedge \ \neg c'.\mathcal{D}_{[i]}$$

Coupling relation is trivially maintained.

□

7.5 Memory Steps

FENCE, INVLPG, SWITCH and WritePTO

Lemma 30 (simulating FENCE, INVLPG, SWITCH, WPTO).

$$I = hd(is_{[i]}) \wedge (FENCE(I) \vee INVLPG(I) \vee SWITCH(I) \vee WPTO(I)) \wedge$$
$$c_{sbh} \sim c \wedge c_{sbh} \stackrel{m}{\Longrightarrow}_i c'_{sbh} \wedge c \stackrel{m}{\Longrightarrow}_i c' \rightarrow c'_{sbh} \sim c'$$

Proof. In order for a step to be scheduled, the SB has to be empty:

$$sb_{[i]} = [].$$

Since the instruction lists are equal in both machines, we know that $hd(c.is_{[i]}) = I$. The coupling for the instruction list is maintained with Lemma 25. For the dirty flag and for the release sets rls_X, where $X \in \{l, s, pt\}$ we have

$$c'.\mathcal{D}_{[i]} = \mathcal{D}'_{[i]} = False$$
$$c'.rls_{X[i]} = rls'_{X[i]} = \emptyset.$$

Since the store buffer of thread i in configuration c'_{sbh} is empty, the coupling for the dirty flag and for the release sets obviously holds.

In case $I =$ **INVLPG** F fi we get for the MMU coupling:

$$c'.mmu_{[i]} = \delta_{flush}(c.mmu_{[i]}, F, fi) = \delta_{flush}(mmu_{[i]}, F, fi) = mmu'_{[i]}.$$

In case $I =$ **WritePTO** v we also get

$$c'.mmu_{[i]} = mmu'_{[i]}$$

with the same argument as in the INVLPG case.

In case $I =$ **Switch** mode for the mode bit coupling we get

$$mode'_{[i]} = c'.mode'_{[i]} = mode$$

All the other parts of the coupling relation can not be possibly broken by a step. □

RMW

Lemma 31 (ownership transfer safe after SB step).

$$inv(c_{sbh}) \wedge safe\_otran(c_{sbh}, i, I) \wedge i \neq j \rightarrow safe\_otran(\delta_{sb}(c_{sbh}, j), i, I)$$

Proof. Let $c'_{sbh} = \delta_{sb}(c_{sbh}, j)$. From the semantics of the SB step we have

$$pt'_{[j]} \cup acq_{pt}(sb'_{[j]}) \subseteq pt_{[j]} \cup acq_{pt}(sb_{[j]})$$
$$\mathcal{O}'_{[j]} \cup acq(sb'_{[j]}) \subseteq \mathcal{O}_{[j]} \cup acq(sb_{[j]}),$$

and conclude the proof. □

Lemma 32 (ownership transfer, Δ_{sb} commute).

$$inv(c_{sbh}) \wedge safe\_otran(c_{sbh}, i, I) \wedge sb_{[i]} = [] \rightarrow$$
$$\Delta_{sb}^{exec}(otran\text{-}sbh(c_{sbh}, i, I)) = otran\text{-}sbh(\Delta_{sb}^{exec}(c_{sbh}), i, I)$$

Proof. We apply Lemma 2 as many times as necessary to reorder all executed SB steps of all threads behind the ownership transfer. After every SB step we use lemmas 1 and 31 to make sure that invariants are maintained after the step and the ownership transfer of instruction I in thread i is still safe. □

Both machines perform the same step.

Lemma 33 (simulating RMW).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge c_{sbh} \stackrel{m}{\Longrightarrow}_i c'_{sbh} \wedge c \stackrel{m}{\Longrightarrow}_i c' \wedge$$
$$hd(is_{[i]}) = \textbf{RMW va t (D, f) r cond } (\mathrm{A}, \mathrm{L}, \mathrm{R}, \mathrm{W}, \mathrm{A_{pt}}, \mathrm{R_{pt}}) \rightarrow c'_{sbh} \sim c'$$

Proof. The coupling for the instruction list, for the dirty flag and for the release sets is maintained with the same arguments as in case of the fence memory step. Since we know that $sb_{[i]}$ is empty, we also get

$$c.is_{[i]} = is_{[i]}$$
$$c.\vartheta_{[i]} = \vartheta_{[i]}.$$

Invariant $tinv3(c_{sbh})$ guarantees that temporary t is fresh. Hence,

$$c.\vartheta_{[i]}(t) = \vartheta_{[i]}(t) = \bot.$$

Let $pa \in (\mathrm{atran}(mmu_{[i]}, va, mode_{[i]}, r))$. Applying Lemma 21 we know that there are no writes to pa in the executed parts of SBs:

$$\forall j.\ \forall k < |exec(sb_{[j]})|.\ \neg(nvW(sb_{[j]}[k]) \wedge sb_{[j]}[k].pa = pa). \tag{11}$$

This implies

$$c.m(pa) = m(pa).$$

Hence, we read the same value into temporary t on both machines and the coupling for temporaries is maintained. Moreover, we can conclude:

$$\mathrm{cond}(c.\vartheta_{[i]}(t \mapsto c.m(pa))) = \mathrm{cond}(\vartheta_{[i]}(t \mapsto m(pa))).$$

In case the RMW test fails and no write is performed all other parts of the coupling invariant can not be possibly broken. Otherwise, we have to show that the coupling for memory and ownership sets is maintained. The coupling for memory cells other than pa is obviously maintained. For pa we get

$$\begin{aligned} c'.m(pa) &= f(c.\vartheta_{[i]}(t \mapsto c.m(pa))) \\ &= f(\vartheta_{[i]}(t \mapsto m(pa))) \\ &= c'_{sbh}.m(pa). \end{aligned}$$

The store buffer of thread i is still empty after the step. (11) guarantees that no other SBs have a write to pa. Hence, the memory coupling for pa is maintained.

From the coupling relation, the safety of the RMW instruction and invariants $sinv4(c_{sbh})$ $oinv4(c_{sbh})$, $pinv1(c_{sbh})$ and $pinv2(c_{sbh})$ we can conclude

$$\begin{aligned}&I.L \subseteq I.A \wedge I.R \subseteq \mathcal{O}_{[i]} \ \wedge I.R_{pt} \subseteq pt_{[i]} \ \wedge\\ &(\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (\mathcal{O}_{[j]} \cup acq(sb_{[j]})) = \emptyset) \ \wedge\\ &(\forall j \neq i.\ (I.A \cup I.A_{pt}) \cap (pt_{[j]} \cup acq_{pt}(sb_{[j]})) = \emptyset).\end{aligned}$$

Hence, we get

$$safe\_otran(c_{sbh}, i, hd(is_{[i]})).$$

Lemma 32 now guarantees that the coupling for the ownership sets of all threads, and for the shared and read only sets is also maintained. □

Read, Write and Ghost

Lemma 34 (simulating R,W,G with vW).

$$\begin{aligned}&c_{sbh} \sim c \wedge I = hd(is_{[i]}) \wedge (R(I) \vee W(I) \vee G(I)) \wedge susp(sb'_{[i]}) \neq [] \ \wedge\\ &c_{sbh} \stackrel{m}{\Longrightarrow}_i c'_{sbh} \rightarrow c'_{sbh} \sim c.\end{aligned}$$

Proof. The coupling for the instruction list is maintained with Lemma 24. If we execute $vW(I)$, then the dirty flag is set and we get

$$\exists k.\ vW(sb'_{[i]}[k]) \Leftrightarrow \mathcal{D}'_{[i]} = True$$

and the coupling for the dirty flag is maintained. For the other parts of the coupling invariant we consider cases:

- $susp(sb_{[i]}) \neq []$. If $R(I)$ then we from $tinv3(c_{sbh})$ we know that $I.t$ is fresh:

  $$\vartheta_{[i]}(t) = \bot.$$

  Hence, we conclude from the coupling relation and from the semantics of a memory step:

  $$\begin{aligned}c.\vartheta_{[i]} &= del\text{-}t(\vartheta_{[i]}, susp(sb_{[i]}))\\ &= del\text{-}t(\vartheta'_{[i]}, susp(sb_{[i]}) \circ I)\\ &= del\text{-}t(\vartheta'_{[i]}, susp(sb'_{[i]})).\end{aligned}$$

  All the other parts of the coupling relation are trivially maintained because

  $$exec(sb_{[i]}) = exec(sb'_{[i]}).$$

- $susp(sb_{[i]}) = []$. This implies $vW(I)$. Since the volatile write is always added to the suspended part of the SB (even if it was empty before) we have

  $$exec(sb_{[i]}) = exec(sb'_{[i]}).$$

  and all parts of the coupling relation are trivially maintained.

□

A memory step is deterministic for a given translated address pa. We introduce a function δ_m to compute the next state of the SB machine after a memory step of thread i. Let $I = hd(is_{[i]})$ then

$$\delta_m(c_{sbh}, i, pa) \equiv c'_{sbh}$$

where:

$$c_{sbh} \xRightarrow{\mathrm{m}}_i c'_{sbh} \wedge pa \in (atran(mmu_{[i]}, I.va, mode_{[i]}, I.r)) \wedge$$
$$(RMW(I) \rightarrow \vartheta'(I.t) = m(pa)) \wedge (W(I) \vee R(I) \rightarrow last(sb'_{[i]}).pa = pa).$$

With this notation we can prove the following lemma.

Lemma 35 (Δ^{exec}_{sb}, δ_m step commute).

$$I = hd(is_{[i]}) \wedge (G(I) \vee nvW(I) \wedge pa = \epsilon(atran(mmu_{[i]}, I.va, mode_{[i]}, I.r))) \wedge$$
$$(\forall k.\ \neg vW(sb_{[i]}[k])) \wedge inv(\delta_m(c_{sbh}, i)) \wedge inv(c_{sbh}) \ \rightarrow$$
$$\Delta^{exec}_{sb}(\delta_m(c_{sbh}, i, pa), i) = \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}, i), i, pa), i) \wedge$$
$$\Delta^{exec}_{sb}(\delta_m(c_{sbh}, i, pa)) = \delta_{sb}(\delta_m(\Delta^{exec}_{sb}(c_{sbh}), i, pa), i)$$

Proof. From the definition of Δ^{exec}_{sb} we can get

$$X \in \{mmu, mode\}.\ X_{[i]} = \Delta^{exec}_{sb}(c_{sbh}, i).X_{[i]} = \Delta^{exec}_{sb}(c_{sbh}).X_{[i]}$$

Thus, we can use the same pa for memory step of c_{sbh}, $\Delta^{exec}_{sb}(c_{sbh}, i)$ and $\Delta^{exec}_{sb}(c_{sbh})$ in thread i. Adding a ghost instruction or a non-volatile write to the SB does not affect other instructions in this SB and does not change the local state of the thread except the state of the SB. Hence, we can first execute old instructions in the SB, then execute a memory step adding the instruction to the empty SB, and finally perform an SB step executing newly added instruction. This concludes the first statement of the lemma.

For the second statement we apply Lemma 6 and conclude

$$\Delta^{exec}_{sb}(\delta_m(c_{sbh}, i, pa)) = \Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(\delta_m(c_{sbh}, i, pa)), i).$$

From the definition of $\Delta^{exec}_{sb[\neq i]}$ and Δ^{exec}_{sb} we can also get

$$X \in \{mmu, mode\}.\ X_{[i]} = \Delta^{exec}_{sb[\neq i]}(c_{sbh}).X_{[i]} = \Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i).X_{[i]}$$

We also can use the same pa for memory step of $\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i)$, c_{sbh} and $\Delta^{exec}_{sb[\neq i]}(c_{sbh})$ and in thread i. Putting an instruction to the store buffer i only affects the component $ts_{[i]}$. Hence, we can reorder it with the store buffer steps of other threads:

$$\Delta^{exec}_{sb}(\Delta^{exec}_{sb[\neq i]}(\delta_m(c_{sbh}, i, pa)), i) = \Delta^{exec}_{sb}(\delta_m(\Delta^{exec}_{sb[\neq i]}(c_{sbh}), i, pa), i).$$

Applying the first statement of the lemma we get

$$\Delta_{sb}^{exec}(\delta_m(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i, pa), i) = \delta_{sb}(\delta_m(\Delta_{sb}^{exec}(\Delta_{sb[\neq i]}^{exec}(c_{sbh}), i), i, pa), i).$$

By applying once again Lemma 6 we conclude the second statement of the lemma. □

Lemma 36 (simulating R,W,G without vW).

$$c_{sbh} \sim c \wedge inv(c_{sbh}) \wedge safe\text{-}reach_d(c) \wedge I = hd(is_{[i]}) \wedge (W(I) \vee G(I) \vee R(I)) \wedge$$
$$susp(sb'_{[i]}) = [] \wedge c_{sbh} \overset{m}{\Longrightarrow}_i c'_{sbh} \rightarrow \exists c'.\ c \overset{m}{\Longrightarrow}_i c' \wedge c'_{sbh} \sim c'$$

Proof. Let $I = hd(is_{[i]})$. Since the suspended part of SB i is empty we have

$$hd(c.is_{[i]}) = I.$$

If I is a read or a write instruction, then from the coupling relation we have

$$\text{atran}(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) = \text{atran}(mmu_{[i]}, I.va, mode_{[i]}, I.r).$$

Hence, we can always execute the instruction from the head of the instruction list of the abstract machine and choose the same translated address as we do in the step of the SB machine . Let c' be configuration of the abstract machine after this step:

$$c \overset{\text{m}}{\Longrightarrow}_i c'.$$

The coupling for the instruction list is maintained with Lemma 24 We now do a case split on the type of the step and consider other parts of the coupling relation which might get broken by this step:

- $G(I)$. Coupling for the ownership sets of threads $j \neq i$ is trivially maintained. Let $X \in \{\mathcal{O}, pt, rls_{pt}\}$. From the coupling invariant and the semantics of the memory step of the abstract and the SB machine we get

$$c'.shared = \delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}), i, pa), i).shared$$
$$c'.ro = \delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}), i, pa), i).ro$$
$$c'.X_{[i]} = \delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}, i), i, pa), i).X_{[i]}.$$

  With Lemma 11 we get $inv(c'_{sbh})$. Applying Lemma 35 we get

$$\delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}), i, pa), i) = \Delta_{sb}^{exec}(c'_{sbh})$$
$$\delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}, i), i, pa), i) = \Delta_{sb}^{exec}(c'_{sbh}, i)$$

  which concludes the coupling for ownership and release pt sets.
  For release local and release shared set, we have to prove

$$I.R \cap c.shared = I.R \cap c_{sbh}.shared \tag{12}$$

From the coupling invariants, we have:

$$c.shared = \Delta_{sb}^{exec}(c_{sbh}).shared$$

With the semantics, we can conclude:

$$\begin{aligned}
&\Delta_{sb}^{exec}(c_{sbh}).shared \subseteq \\
&c_{sbh}.shared \bigcup_{\forall j}(rels(exec(sb_{[j]})) \cup rels_{pt}(exec(sb_{[j]}))), \\
&\Delta_{sb}^{exec}(c_{sbh}).shared \supseteq \\
&c_{sbh}.shared \setminus (\bigcup_{\forall j}(acq(exec(sb_{[j]})) \cup acq_{pt}(exec(sb_{[j]})))).
\end{aligned}$$

With $sinv4(c_{sbh})$, $oinv4(c_{sbh})$ and the semantics we can conclude:

$$\begin{aligned}
I.R \cap c.shared \subseteq I.R \cap c_{sbh}.shared, \\
I.R \cap c.shared \supseteq I.R \cap c_{sbh}.shared.
\end{aligned}$$

which concludes (12).

– $nvW(I)$. Let $I =$ **Write** False a (D, f) r annot and pa be the translated address chosen in both the SB and the abstract machine. From the coupling invariant and the semantics of the memory step of the abstract and the SB machine we get

$$c'.m = \delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}), i, pa), i).m.$$

With Lemma 12 we get $inv(c'_{sbh})$. Applying Lemma 35 we conclude

$$\delta_{sb}(\delta_m(\Delta_{sb}^{exec}(c_{sbh}), i, pa), i) = \Delta_{sb}^{exec}(c'_{sbh}).$$

– $R(I)$. Let pa be the translated address chosen in both the SB and the abstract machine. Lemma 21 guarantees that there are no writes to pa in the executed parts of store buffers other than i. With the coupling relation for memory we conclude

$$c.m(pa) = fwd(sb_{[i]}, m, pa).$$

From the coupling for temporaries we get

$$c.\vartheta_{[i]} = \vartheta_{[i]}.$$

Hence, we conclude

$$\begin{aligned}
c'.\vartheta_{[i]} &= c.\vartheta_{[i]}(t \mapsto c.m(pa)) \\
&= \vartheta_{[i]}(t \mapsto fwd(sb_{[i]}, m, pa)) \\
&= \vartheta'_{[i]}.
\end{aligned}$$

□

8 Proving Safety of the Delayed Release

So far we have used safety of the delayed release of the virtual machine to prove SB reduction theorem. Now we have to show that if all possible executions of the virtual machine satisfy the regular safety, then they also satisfy safety of the delayed release.

8.1 Intuition

If a trace satisfies regular safety, but does not satisfy safety of the delayed release, then the safety violation is due to a clash with release sets of some thread i. This clash happen between (i) an instruction in the head of the instruction list of thread $j \neq i$, which can access or acquire an address from release sets of thread i, or (ii) an MMU of thread $j \neq i$ which can access an address from release sets of thread i, or (iii) an MMU of thread i which can access an address from its own local release set. We do the prove by contradiction: we show that if some execution does not satisfy safety of the delayed release, then there exists another execution, which does not satisfy regular safety. We can obtain such an execution by "undoing" steps of thread i until we reach a point when the conflicting address was released. The steps of thread i which are removed can only be program steps or memory steps executing ghost instructions, reads and non-volatile writes (since all the other instructions are clearing the release sets). All these steps do not have any affect on executions of other threads. Hence, after removing these steps of thread i we can continue our execution until we get the safety violation. Since in the new execution thread i has not put the conflicting address to the release thread yet, it will be either in the owns or in the PT set of thread i. Thus, we will get violation of the regular safety. We also might end up in a situation when we encounter violation of the regular safety earlier in the new execution. This is also fine, since we assume all traces to satisfy regular safety. Note, that we are not removing the MMU steps of thread i, because MMUs are allowed to write the shared memory and can possibly affect the execution flow of other threads. Below we consider a few examples.

Example 1. Let address pa be in the ownership set of thread i:

$$pa \in c.\mathcal{O}_{[i]}.$$

Let thread i execute a volatile write, a ghost release of address pa, a volatile read and an MMU write. After that thread j performs a volatile write acquiring address pa (Fig. 14a).

This behaviour satisfies regular safety, because at the time when thread j acquires pa it is not present in the ownership set of thread i. Yet, it is present in the release set of thread i, which means that safety of the delayed release is not satisfied. To rule out this situation we consider another trace, where we "undo" the two last memory steps of thread i (Fig. 14b).

The read and the ghost operation of thread i do not affect execution of thread j. Moreover, since we allow only volatile writes to page tables, these steps also

thread i	thread j
Write True va (D, f) r annot	—
Ghost (A, L, {pa}, W, A_{pt}, R_{pt})	—
Read True va t r	—
MMU Write pa′	—
—	**Write** True va (D, f) r ({pa}, L′, R′, W′, A'_{pt}, R'_{pt})

(a) Violation of safety of the delayed release.

thread i	thread j
Write True va (D, f) r annot	—
MMU Write pa′	—
—	**Write** True va (D, f) r ({pa}, L′, R′, W′, A'_{pt}, R'_{pt})

(b) Violation of regular safety.

Fig. 14: Ruling out safety of the delayed release violation: Example 1.

thread i
Write True va (D, f) r annot
Ghost (A, L, {pa}, W, A_{pt}, R_{pt})
MMU Read pa

(a) Violation of safety of the delayed release.

thread i
Write True va (D, f) r annot
MMU Read pa
—

(b) Violation of regular safety.

Fig. 15: Ruling out safety of the delayed release violation: Example 2.

cannot affect the MMU steps of thread i. Hence, we can simply skip execution of those steps, execute the MMU write immediately after the volatile write of thread i and then perform the step of thread j.

Note, that we can not simply reorder the removed steps of thread i after the step of thread j, because the volatile read in thread i might read the value written by the volatile write of thread j. However, we don't need to execute the removed steps of thread i to get a safety violation: since address *pa* is present in the ownership set of thread i, attempt to acquire it by thread j already violates the regular safety of the virtual machine.

Example 2. Let address *pa* again be in the ownership set of thread i:

$$pa \in c.\mathcal{O}_{[i]}.$$

Let thread i execute a volatile write and a ghost release of address *pa*. After that MMU of thread i attempts to perform a read from *pa*. (Fig. 15a).

This behaviour again satisfies regular safety, because at the time when MMU performs a read address *pa* is shared and is not owned by any thread. Yet, it is present in the release set of thread i, which means that safety of the delayed release is not satisfied. To rule out this situation we "undo" the last memory step of thread i (Fig. 15b) and get a trace which violates regular safety.

thread i	thread j
Write True va (D, f) r annot	—
Ghost $(\mathrm{A}, \mathrm{L}, \{\mathrm{pa}\}, \mathrm{W}, \mathrm{A}_{\mathrm{pt}}, \mathrm{R}_{\mathrm{pt}})$	—
Ghost $(\mathrm{A}, \mathrm{L}, \mathrm{R}, \mathrm{W}, \mathrm{A}_{\mathrm{pt}}, \{\mathrm{pa}'\})$	—
—	**Write** True va (D, f) r $(\{\mathrm{pa}\}, \mathrm{L}', \mathrm{R}', \mathrm{W}', \{\mathrm{pa}'\}, \mathrm{R}'_{\mathrm{pt}})$

(a) Violation of safety of the delayed release.

thread i	thread j
Write True va (D, f) r annot	—
Ghost $(\mathrm{A}, \mathrm{L}, \{\mathrm{pa}\}, \mathrm{W}, \mathrm{A}_{\mathrm{pt}}, \mathrm{R}_{\mathrm{pt}})$	—
—	**Write** True va (D, f) r $(\{\mathrm{pa}\}, \mathrm{L}', \mathrm{R}', \mathrm{W}', \{\mathrm{pa}'\}, \mathrm{R}'_{\mathrm{pt}})$

(b) Violation of regular safety.

Fig. 16: Ruling out safety of the delayed release violation: Example 3.

Example 3. Let address pa again be in the ownership set of thread i and address pa' be in the PT set of thread i:

$$pa \in c.\mathcal{O}_{[i]} \quad \text{and} \quad pa' \in c.pt_{[i]}.$$

Let thread i execute a volatile write, a ghost release of address pa, and a ghost release of address pa'. After that thread j attempts to acquire both addresses in a single volatile write (Fig. 16a). Here we have two violations of the safety of the delayed release in a single instruction. Suppose we take the violation for address pa and try to remove all steps of thread i starting from the ghost instruction which releases pa. But after we remove the first instruction (i.e., the ghost instruction which releases pa') we already get violation of regular safety with respect to pa' (Fig. 16b). Hence, we do not need to remove any more steps, because any unsafe trace can be used to rule out the situation.

8.2 "Undoing" a Step

We define a simulation relation, which is supposed to hold between the states of the original execution and the states of the execution where a step of thread i has not been performed:

$$\begin{aligned} simd(c, d, i) \equiv \forall j \neq i.\ & c.ts_{[j]} = d.ts_{[j]} \wedge \\ & c.mmu_{[i]} = d.mmu_{[i]} \wedge c.mode_{[i]} = d.mode_{[i]} \wedge \\ & c.rls_{l[i]} \subseteq d.rls_{l[i]} \cup (d.\mathcal{O}_{[i]} \setminus d.shared) \wedge \\ & c.rls_{s[i]} \subseteq d.rls_{s[i]} \cup d.\mathcal{O}_{[i]} \wedge c.rls_{pt[i]} \subseteq d.rls_{pt[i]} \cup d.pt_{[i]} \wedge \\ & \forall a.\ a \notin d.\mathcal{O}_{[i]} \vee a \in d.shared.\ c.m(a) = d.m(a). \end{aligned}$$

Lemma 37 ensures that we can "undo" a step of thread i. This means that relation $simd$ holds between the states before and after a step of thread i, if this step is a program step or a memory step executing a ghost instruction, a read or a non-volatile write.

Lemma 37 (undoing a step).

$$(c \overset{p}{\Longrightarrow}_i c' \vee c \overset{m}{\Longrightarrow}_i c' \wedge I = hd(c.is_{[i]}) \wedge (G(I) \vee nvW(I) \vee R(I))) \wedge \\ safe\text{-}state(c) \rightarrow simd(c', c, i)$$

Proof. Since the step of thread i can not affect the local configuration of threads $j \neq i$ we obviously get

$$c'.ts_{[j]} = c.ts_{[j]}.$$

If a step of thread i is a memory step executing a ghost instruction I, then we have

$$\begin{aligned} c'.rls_{l[i]} &= c.rls_{l[i]} \cup (I.R \setminus c.shared) \\ c'.rls_{s[i]} &= c.rls_{s[i]} \cup (I.R \cap c.shared) \\ c'.rls_{pt[i]} &= c.rls_{pt[i]} \cup I.R_{pt}. \end{aligned}$$

From $safe\text{-}state(c)$ we have

$$I.R \subseteq c.\mathcal{O}_{[i]} \quad \text{and} \quad I.R_{pt} \subseteq c.pt_{[i]}.$$

Hence, we get

$$\begin{aligned} c'.rls_{l[i]} &\subseteq c.rls_{l[i]} \cup (c.\mathcal{O}_{[i]} \setminus c.shared) \wedge \\ c'.rls_{s[i]} &\subseteq c.rls_{s[i]} \cup c.\mathcal{O}_{[i]} \wedge \\ c'.rls_{pt[i]} &\subseteq c.rls_{pt[i]} \cup c.pt_{[i]}. \end{aligned}$$

If a step of thread i is a memory step executing a non-volatile write instruction I, then from the safety of configuration c we have for all physical addresses pa:

$$pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}, I.r) \rightarrow pa \in c.\mathcal{O}_{[i]} \setminus c.shared.$$

This implies

$$\forall a.\ a \notin c.\mathcal{O}_{[i]} \vee a \in c.shared.\ c'.m(a) = c.m(a)$$

and concludes $simd(c', c, i)$. □

The following lemma guarantees that if $simd(c, d, i)$ holds and we perform a step from configuration c which is not a program or memory step of thread i, then we can also perform a step from configuration d, such that the simulation relation is maintained after the step.

Lemma 38 (simd maintained).

$$simd(c, d, i) \wedge disjoint\text{-}osets(d) \wedge safe\text{-}state(d) \wedge \\ (c \overset{mu}{\Longrightarrow}_i c' \vee c \Longrightarrow_j c' \wedge j \neq i) \rightarrow \exists d'.\ d \Longrightarrow d' \wedge simd(c', d', i)$$

Proof. We split cases on the kind of a step from c to c'.

- A step from c to c' is a memory step of thread $j \neq i$. From $simd(c, d, i)$ we get
$$c.ts_{[j]} = d.ts_{[j]}.$$
Hence, we can execute the (same) first instruction I of thread j with the same address translation in both machines. Let d' be the configuration after we execute this instruction from configuration d:
$$d \xRightarrow{\mathrm{m}}_j d'.$$
If this instruction is doing a read (either $R(I)$ or $RMW(I)$) from address pa then from $safe\text{-}state(d)$ and $disjoint\text{-}osets(d)$ we conclude:
$$pa \notin d.\mathcal{O}_{[i]} \vee pa \in d.shared.$$
Hence, $simd(c, d, i)$ guarantees that we are reading the same value in both machines. This implies
$$c'.ts_{[j]} = d'.ts_{[j]}.$$
If instruction I is performing ownership transfer or write to memory ($W(I)$ or $G(I)$ or $RMW(I)$) then from $safe\text{-}state(d)$ we get
$$I.R \subseteq d.\mathcal{O}_{[j]} \wedge I.R_{pt} \subseteq d.pt_{[j]}.$$
With $disjoint\text{-}osets(d)$ we have
$$I.R \cap d.\mathcal{O}_{[i]} = I.R_{pt} \cap d.\mathcal{O}_{[i]} = \emptyset.$$
Configuration of thread i is not changed during a step. Hence,
$$d.\mathcal{O}_{[i]} \setminus d.shared \subseteq d'.\mathcal{O}_{[i]} \setminus d'.shared,$$
which together with $simd(c, d, i)$ implies
$$c'.rls_{l[i]} \subseteq d'.rls_{l[i]} \cup (d'.\mathcal{O}_{[i]} \setminus d'.shared).$$
If instruction I is writing the memory, then we are writing the same value for both configurations. Moreover, we observe that
$$\forall a.\ a \notin d.\mathcal{O}_{[i]} \vee a \in d.shared \leftrightarrow a \notin d'.\mathcal{O}_{[i]} \vee a \in d'.shared$$
which concludes $simd(c', d', i)$.
- A step from c to c' is a program step of thread $j \neq i$. From $simd(c, d, i)$ we get
$$c.ts_{[j]} = d.ts_{[j]},$$
which means that we can perform the same program step for both configurations and get $simd(c', d', i)$.

– A step from c to c' is an MMU step of thread i accessing address pa. Thus, it holds
$$\mathit{can\text{-}access}(c.\mathit{mmu}_{[i]}, pa).$$
From $\mathit{simd}(c, d, i)$ we know that
$$c.\mathit{mmu}_{[i]} = d.\mathit{mmu}_{[i]} \quad \text{and} \quad c.\mathit{mode}_{[i]} = d.\mathit{mode}_{[i]}.$$
Hence, we have
$$\mathit{can\text{-}access}(d.\mathit{mmu}_{[i]}, pa).$$
Safety of configuration d ensures
$$pa \notin d.\mathcal{O}_{[i]}.$$
Hence, from $\mathit{simd}(c, d, i)$ we get
$$c.m(pa) = d.m(pa).$$
This implies that we can perform the same kind of MMU step from both configurations resulting with the same MMU configuration:
$$c'.\mathit{mmu}_{[i]} = d'.\mathit{mmu}_{[i]}.$$
If MMU step is writing the memory, then we write the same value for both configurations and get
$$c'.m(pa) = d'.m(pa),$$
which concludes $\mathit{simd}(c', d', i)$.
– A step from c to c' is an MMU step of thread $j \neq i$ to address pa. From $\mathit{simd}(c, d, i)$ we get
$$c.\mathit{ts}_{[j]} = d.\mathit{ts}_{[j]}.$$
We easily conclude $\mathit{simd}(c', d', i)$ the same way as in case of the MMU step of thread i.

□

The following lemma guarantees that we can continue execution of the machine after we "undo" a step of thread i.

Lemma 39 (simd computation).

$$\begin{aligned}&n > 0 \wedge c^0 \Longrightarrow^n c^n \wedge \forall k < n. \neg(c^k \overset{p,m}{\Longrightarrow}_i c^{k+1} \vee c^k \overset{pf}{\Longrightarrow}_i c^{k+1}) \wedge \\ &\mathit{safe\text{-}reach}(d^0) \wedge \mathit{disjoint\text{-}osets}(d^0) \wedge \mathit{simd}(c^0, d^0, i) \;\rightarrow \\ &\exists d^n.\ d^0 \Longrightarrow^n d^n \wedge \mathit{simd}(c^n, d^n, i)\end{aligned}$$

Proof. We will prove an inductive statement, which trivially implies the post-condition of the lemma:
$$\forall l \leq n.\ \exists d^l.\ \mathit{simd}(c^l, d^l, i) \wedge (l > 0 \rightarrow d^0 \Longrightarrow^l d^l).$$

Proof by induction on l. For induction base $l = 0$ we obviously take $x = 0$ and have from the preconditions:

$$simd(c^0, d^0, i).$$

For the induction step $l \to l + 1$ we have from the induction hypothesis

$$\exists d^l.\ simd(c^l, d^l, i) \wedge (l > 0 \to d^0 \Rightarrow^l d^l).$$

Step l in the original computation is either an MMU step of thread i or is an arbitrary step of thread $j \neq i$. With Lemma 8 we get

$$disjoint\text{-}osets(d^l).$$

Hence, we can apply Lemma 38 to find configuration d^{l+1}, where

$$d^l \Rightarrow d^{l+1} \quad \text{and} \quad simd(c^{l+1}, d^{l+1}, i).$$

□

8.3 Reconstructing Safety Violation

The following predicate denotes that in configuration c there exists a safety violation due to a clash with release sets of thread j. Let $\vartheta' = c.\vartheta_{[i]}(I.t \mapsto c.m(pa))$ and $I = hd(c.is_{[i]})$, then

$$\begin{aligned}
&unsafe\text{-}release(c, j) = \\
&\quad (\exists i \neq j.\ \exists pa \in atran(c.mmu_{[i]}, I.va, c.mode_{[i]}).\\
&\qquad (vR(I) \vee (RMW(I) \wedge \neg I.cond(\vartheta')) \wedge pa \in c.rls_{l[j]} \cup c.rls_{pt[j]}) \vee\\
&\qquad (nvR(I) \vee W(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \wedge pa \in c.rls_{[j]}) \vee\\
&\qquad (vW(I) \vee G(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \wedge (I.A \cup I.A_{pt}) \cap c.rls_{[j]} \neq \emptyset)) \vee\\
&\quad (\exists pa, i.\ can\text{-}access(c.mmu_{[i]}, pa) \wedge (a \in c.rls_{[j]} \wedge i \neq j \vee i = j \wedge a \in c.rls_{l[i]})).
\end{aligned}$$

Lemma 40 ensures that we can reconstruct a safety violation after we "undo" a step of thread i and execute all the remaining steps in such a way, that relation *simd* holds between the faulty state of the original computation and the end state of the new computation.

Lemma 40 (reconstructing safety violation).

$$simd(c, d, i) \wedge unsafe\text{-}release(c, i) \wedge disjoint\text{-}osets(d) \to \neg safe\text{-}state_d(d)$$

Proof. From $simd(c, d, i)$ we know that

$$\begin{aligned}
\forall m \neq i.\ c.ts_{[m]} &= d.ts_{[m]} \qquad (13)\\
c.mmu_{[i]} &= d.mmu_{[i]}\\
c.mode_{[i]} &= d.mode_{[i]}\\
c.rls_{l[i]} &\subseteq d.rls_{l[i]} \cup (d.\mathcal{O}_{[i]} \setminus d.shared)\\
c.rls_{s[i]} &\subseteq d.rls_{s[i]} \cup d.\mathcal{O}_{[i]}\\
c.rls_{pt[i]} &\subseteq d.rls_{pt[i]} \cup d.pt_{[i]}.
\end{aligned}$$

We split cases:

– MMU safety violation for address pa in thread $m \neq i$:

$$\mathit{can\text{-}access}(c.mmu_{[m]}, pa) \wedge pa \in c.rls_{[i]}$$

From (13) we get

$$\mathit{can\text{-}access}(d.mmu_{[m]}, pa) \wedge pa \in d.\ rls_{[i]} \cup d.\mathcal{O}_{[i]} \cup d.pt_{[i]}.$$

If $pa \in d.\ rls_{[i]} \cup d.\mathcal{O}_{[i]}$, then configuration d does not satisfy safety of the delayed release and we are done. If $pa \in d.pt_{[i]}$, then $\mathit{disjoint\text{-}osets}(d)$ ensures

$$pa \notin d.ts_{[m]}.pt \cup d.shared,$$

which also gives safety violation and concludes the proof.

– MMU safety violation for address pa in thread i:

$$\mathit{can\text{-}access}(c.mmu_{[i]}, pa) \wedge pa \in c.rls_{l[i]}.$$

As in the previous case we use (13) to get

$$\mathit{can\text{-}access}(d.mmu_{[i]}, pa) \wedge pa \in d.rls_{l[i]} \cup d.\mathcal{O}_{[i]},$$

which violates safety and concludes the proof.

– Instruction safety violation in thread $m \neq i$. Let $I = hd(c.is_{[m]})$ be the faulty instruction. Safety violation can be caused either by a physical address of the instruction being present in the release sets of thread i or by a clash between the acquire sets of instruction I and the release sets of thread i. For the first class of violations let

$$pa \in atran(c.mmu_{[m]}, I.va, c.mode_{[m]}, I.r)$$

be the faulty address. From (13) we immediately get

$$pa \in atran(d.mmu_{[m]}, I.va, d.mode_{[m]}, I.r).$$

If instruction I is a read or an RMW instruction, then from $safe\text{-}state(d)$ and $\mathit{disjoint\text{-}osets}(d)$ we conclude:

$$pa \notin d.\mathcal{O}_{[i]} \vee pa \in d.shared.$$

Hence, $simd(c, d, i)$ guarantees that the result of a read in configurations c and d is the same:

$$c.m(pa) = d.m(pa).$$

We consider sub-cases, where $\vartheta' = c.\vartheta_{[m]}(I.t \mapsto c.m(pa))$:

- $(vR(I) \vee (RMW(I) \wedge \neg I.cond(\vartheta')) \wedge pa \in c.rls_{l[i]} \cup c.rls_{pt[i]})$. From (13) we get

$$pa \in d.rls_{l[i]} \cup (d.\mathcal{O}_{[i]} \setminus d.shared) \cup d.rls_{pt[i]} \cup d.pt_{[i]}.$$

If pa is present in one of the release sets:

$$pa \in d.rls_{l[i]} \cup d.rls_{pt[i]},$$

then configuration d is unsafe and we are done. If

$$pa \in (d.\mathcal{O}_{[i]} \setminus d.shared) \cup d.pt_{[i]}$$

then with $disjoint\text{-}osets(d)$ we get

$$pa \notin d.\mathcal{O}_{[m]} \cup d.shared \cup d.pt_{[m]}$$

and conclude the proof.

- $(nvR(I) \vee nvW(I)) \wedge pa \in c.rls_{[i]})$. From (13) we get

$$pa \in d.rls_{[i]} \cup d.\mathcal{O}_{[i]} \cup d.pt_{[i]}.$$

With $disjoint\text{-}osets(d)$ we get

$$pa \in d.rls_{[i]} \vee pa \notin d.\mathcal{O}_{[m]} \cup d.ro \cup d.pt_{[m]}$$

and conclude the proof.

- $(vW(I) \vee (RMW(I) \wedge I.cond(\vartheta')) \wedge pa \in c.rls_{[i]})$. From (13) we get

$$pa \in d.rls_{[i]} \cup d.\mathcal{O}_{[i]} \cup d.pt_{[i]},$$

which already gives us safety violation.

□

8.4 Simulation Theorem

In the intuitive explanations which we gave in the beginning of this section we are constructing a new trace by undoing all steps of the conflicting thread until we reach a point when the conflicting address is being released. Nevertheless, we do here a simpler proof. We state an induction hypothesis that all traces up to length n satisfy safety of the delayed release. On the induction step we prove by contradiction and assume there exists a trace of length $n+1$ which does not satisfy safety of the delayed release. We undo only a single step of the conflicting thread i.e., the last memory or program step. We then continue the execution and show that the shorter trace will also violate safety of the delayed release (we assume the regular safety to be always satisfied by all possible traces). Since existence of such a trace contradicts to our induction hypothesis, we conclude that all traces of length $n+1$ satisfy safety of the delayed release.

Lemma 41 (safety ind).

$$initial(c) \wedge safe\text{-}reach(c) \rightarrow \forall k \leq n.\ safe\text{-}reach_d(c, k)$$

Proof. By induction on n. For case $n = 0$ the statement trivially holds since all release sets are empty. For the induction step $n \to n+1$ we do a prove by contradiction. Assume

$$safe\text{-}reach(c) \wedge \neg(\forall k \leq n+1.\ safe\text{-}reach_d(c,k)).$$

Our induction hypothesis guarantees that all configurations up to step n are safe:

$$\forall k \leq n.\ safe\text{-}reach_d(c,k).$$

Hence, there must exist a trace with $n+1$ steps starting from configuration c, where the first n steps are safe and the last step is not safe. We denote the states in this computation by $c^0, \ldots, c^n, c^{n+1}$, where $c^0 = c$, $c^i \Longrightarrow c^{i+1}$ and

$$\neg safe\text{-}state_d(c^{n+1}) \wedge \forall k \leq n.\ safe\text{-}state_d(c^k).$$

From the precondition of the function we know that state c^{n+1} satisfies regular safety of the virtual machine:

$$safe\text{-}conf_{abs}(c^{n+1}).$$

Hence, the safety violation is due to a clash with release sets of some thread i:

$$unsafe\text{-}release(c^{n+1}, i).$$

In this case we aim at "undoing" the last program or memory step of thread i and arguing that a (shorter) trace without this steps would still be unsafe, which contradicts to our induction hypothesis. Note, that after the last program or memory step of thread i there can be no page fault steps of thread i, since this step would empty the release sets.

Let k be the state before the last program or memory step of thread i in the computation:

$$c^k \overset{\text{p,m}}{\Longrightarrow}_i c^{k+1} \wedge \forall m \in [k+1 : n-1].\ \neg(c^m \overset{\text{p,m}}{\Longrightarrow}_i c^{m+1} \vee c^m \overset{\text{pf}}{\Longrightarrow}_i c^{m+1}).$$

If this step is a memory step, then it can execute a ghost instruction, a read or a non-volatile write, since all other instructions empty the release sets. Hence, we can apply Lemma 37 to get

$$simd(c^{k+1}, c^k, i).$$

From $initial(c^0)$ we have

$$disjoint\text{-}osets(c^0).$$

With Lemma 8 we get

$$disjoint\text{-}osets(c^k).$$

From $safe\text{-}reach(c^0)$ we have $safe\text{-}reach(c^k)$. We now split cases:

– if $k = n$ then we are removing the last step in our execution sequence and we have

$$simd(c^{n+1}, c^n, i).$$

Hence, we apply Lemma 40 to reconstruct the safety violation in configuration c^n and get

$$\neg safe\text{-}state_d(c^n),$$

which contradict to our induction hypothesis.

– if $k < n$ then we apply Lemma 39 (we instantiate $d^0 = c^k$, $c^0 = c^{k+1}$, $n = (n - k)$) and get

$$\exists d^{n-k}.\ c^k \Rightarrow^n d^{n-k} \wedge simd(c^{n+1}, d^{n-k}, i)$$

Since $k + (n - k) < n + 1$, the constructed sequence is shorter than the original one. With Lemma 8 we get $disjoint\text{-}osets(d^{n-k})$. Hence, we apply Lemma 40 to reconstruct the safety violation in configuration d^{n-k} and get

$$\neg safe\text{-}state_d(d^{n-k}),$$

which contradicts to our induction hypothesis.

□

The proof of Theorem 3 now simply follows from Lemma 41.

9 Conclusion

We presented a programming discipline for concurrent programs running in translated mode and racing with MMUs. Our reduction theorem guarantees, that if such program satisfies the safety conditions in a sequentially consistent environment, then execution of this program under TSO also preserves sequential consistency. As a result, one can derive properties for TSO executions by verifying programs in a sequentially consistent model of executions. Our main motivation for extending the store buffer reduction theorem with MMUs was driven by the results obtained during the Verisoft XT project [10], which aimed at the formal verification of the Microsoft Hyper-V hypervisor in VCC. When the project ended in 2010 many crucial portions of the hypervisor code were verified. Yet, the overall theory of the multi-core hypervisor verification was far from being fully completed, even on paper. Since then, many pieces of the theory have been worked out [4]. The work presented here is intended to close another open gap in that theory.

Future work. Currently we define the safety properties with respect to the ownership for physical addresses. However, in the presence of address translation we would like to state safety properties for virtual addresses and then automatically derive safety for translated physical addresses. Extending our theory with this machinery remains as a future work. Another extension which is needed in the

context of the multi-core hypervisor verification theory is the support for memory accesses of different width. This requires introduction of byte-write signals to memory operations and byte-wise ownership. Finally, we want to integrate our theory as a part of the model stack of computational models for hypervisor verification. A simplified version of this stack with a brute-force store buffer reduction (i.e. flush after every access to shared data) is given in [7].

References

1. Advanced Micro Devices: AMD64 Architecture Programmer's Manual Volume 2: System Programming, 3.19 edn. (Sep 2011)
2. Alkassar, E., Cohen, E., Kovalev, M., Paul, W.: Verification of TLB virtualization implemented in C. In: Joshi, R., Müller, P., Podelski, A. (eds.) Verified Software: Theories, Tools, Experiments, Lecture Notes in Computer Science, vol. 7152, pp. 209–224. Springer Berlin / Heidelberg (2012)
3. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2009). Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer, Munich, Germany (2009)
4. Cohen, E., Paul, W., Schmaltz, S.: Theory of multi core hypervisor verification. In: van Emde Boas et al., P. (ed.) Invited paper. To appear on SOFSEM 2013, Theory and Practice of Computer Science. LNCS, Springer (2013)
5. Cohen, E., Schirmer, B.: From total store order to sequential consistency: A practical reduction theorem. In: Kaufmann, M., Paulson, L., Norrish, M. (eds.) Interactive Theorem Proving (ITP 2010). Lecture Notes in Computer Science, vol. 6172, pp. 403–418. Springer, Edinburgh, UK (Jul 2010)
6. Hillebrand, M., Leinenbach, D.: Formal verification of a reader-writer lock implementation in C. In: 4th International Workshop on Systems Software Verification (SSV09). Electronic Notes in Theoretical Computer Science, vol. 254, pp. 123–141. Elsevier Science B. V. (2009)
7. Kovalev, M.: TLB Virtualization in the Context of Hypervisor Verification. Ph.D. thesis, Saarland University, Saarbrcken (2013), `http://www-wjp.cs.uni-saarland.de/publikationen/Kov13.pdf`
8. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (Sep 1979), `http://dx.doi.org/10.1109/TC.1979.1675439`
9. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. Communications of the ACM 53(7), 89–97 (2010)
10. Verisoft XT Consortium: The Verisoft XT Project. `http://www.verisoftxt.de` (2007-2010)